UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Characterizing Time Varying Program Behavior for Efficient Simulation

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Erez Perelman

Committee in charge:

Professor Brad Calder, Chair
Professor Tim Sherwood
Professor Paul Siegel
Professor Dean Tullsen
Professor Geoff Voelker

2007

The dissertation of Erez Perelman is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2007

*The first principle is that you must not fool yourself–*

*and you are the easiest person to fool.*

– Richard Feynman

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# ACKNOWLEDGMENTS

The text of Chapter III and Section IV.A are in part a reprint of the material from the paper, *Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood, "Using Machine Learning to Guide Architecture Simulation"*, in the Journal of Machine Learning Research, 2006. The dissertation author was the secondary researcher and author and the co-authors involved in the publication [23] directed, supervised, and assisted in the research which forms the basis for that material.

The text of Chapter IV is in part a reprint of the material from the paper, *Erez Perelman, Greg Hamerly, and Brad Calder, "Picking Statistically Valid and Early Simulation Points"*, in the proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2003) September 2003, New Orleans, Louisiana. The dissertation author was the primary researcher and author and the co-authors involved in the publication [52] directed, supervised, and assisted in the research which forms the basis for that material.

The text of Chapter V is in part a reprint of the material from the paper, *Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, and Brad Calder, "Cross Binary Simulation Points"*, in the proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS 2007) April 2007, San Jose, California. The dissertation author was the primary researcher and author and the co-authors involved in the publication [54] directed, supervised, and assisted in the research which forms the basis for that material.

The text of Section V.A and Section V.A.3 are in part a reprint of the material from the paper, *Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder, "Motivation for Variable Length Intervals and Hierarchical Phase Behavior"*, in the proceedings of the IEEE International Sym-

posium on Performance Analysis of Systems and Software (ISPASS 2005) March 2005, Austin, Texas. The dissertation author was the secondary researcher and author and the co-authors involved in the publication [33] directed, supervised, and assisted in the research which forms the basis for that material.

The text of Chapter VI is in part a reprint of the material from the paper, *Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder, and Carole Dulong, "Detecting Phases in Parallel Applications on Shared Memory Architectures"*, in the proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006) April 2006, Rhodes Island, Greece. The dissertation author was the primary researcher and author and the co-authors involved in the publication [55] directed, supervised, and assisted in the research which forms the basis for that material.

## VITA

| | |
|---|---|
| January 17, 1979 | Born, Be'er-Sheva, Israel |
| 1988 | Moved to USA |
| 2001 | Bachelor of Science in Computer Science, University of California, San Diego |
| 2003 | Internship, Microsoft Research, Redmond, WA |
| 2004 | Internship, Intel, Santa Clara, CA |
| 2007 | Doctor of Philosophy, University of California, San Diego |

## PUBLICATIONS

Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, and Brad Calder. Cross Binary Simulation Points. *In the proceedings of the International Symposium on Performance Analysis of Systems and Software* (ISPASS 2007) April 2007. San Jose, California

Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood, Using Machine Learning to Guide Architecture Simulation, Journal of Machine Learning Research, 2006.

Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder, and Carole Dulong. Detecting Phases in Parallel Applications on Shared Memory Architectures. *In the proceedings of the IEEE International Parallel and Distributed Processing Symposium* (IPDPS 2006) April 2006. Rhodes Island, Greece

Greg Hamerly, Erez Perelman, and Brad Calder. Comparing Multinomial and K-Means Clustering for SimPoint. *In the proceedings of the International Symposium on Performance Analysis of Systems and Software* (ISPASS 2006) March 2006. Austin, Texas

Jeremy Lau, Erez Perelman, and Brad Calder. Selecting Software Phase Markers with Code Structure Analysis. *In the proceedings of the International Symposium on Code Generation and Optimization* (CGO 2006) March 2006. Manhattan, New York

Erez Perelman, Trishul Chilimbi, and Brad Calder. Variational Path Profiling. *In the proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT 2005) September 2005. Saint Louis, Missouri

Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder, SimPoint 3.0: Faster and More Flexible Program Analysis, Workshop on Modeling, Benchmarking and Simulation, June 2005.

Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, and Brad Calder. The Strong Correlation between Code Signatures and Performance. *In the proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS 2005) March 2005. Austin, Texas

Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. *In the proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS 2005) March 2005. Austin, Texas

Greg Hamerly, Erez Perelman, and Brad Calder, How to Use SimPoint to Pick Simulation Points, ACM SIGMETRICS Performance Evaluation Review, Volume 31(4), March 2004.

Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and Exploiting Program Phases. IEEE Micro: Micro's Top Picks from Computer Architecture Conferences, December 2003.

Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. *In the proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT 2003) September 2003. New Orleans, Louisiana

Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder, Using SimPoint for Accurate and Efficient Simulation, ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems, June 2003.

Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior. *In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2002), October 2002. San Jose, California

Timothy Sherwood, Erez Perelman and Brad Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. *In the proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT 2001) Sept, 2001. Barcelona, Spain

## ABSTRACT OF THE DISSERTATION

Characterizing Time Varying Program Behavior for Efficient Simulation

by

Erez Perelman
Doctor of Philosophy in Computer Science
University of California, San Diego, 2007
Professor Bradley Calder, Chair

An essential step in designing a new computer architecture is the careful examination of different design options. It is critical that computer architects have efficient means by which they may estimate the impact of various design options on the overall machine. This task is complicated by the fact that different programs, and even different parts of the *same* program, may have distinct behaviors that interact with the hardware in different ways. Researchers use very detailed simulators to estimate processor performance, which models every cycle of an executing program. Unfortunately, simulating every cycle of a single benchmark program takes on the order of months to complete.

To address this problem we develop analysis techniques for characterizing the time varying program behavior. Using data clustering algorithms from machine learning to automatically find repetitive patterns in a program's execution we can avoid simulating the same behavior many times. By simulating one representative of each repetitive behavior pattern, simulation time can be reduced to hours instead of months for standard benchmark programs, with very little cost in terms of accuracy.

This dissertation describes this important problem and the tool we created, called SimPoint, to automatically find simulation points in programs. Additionally, we describe data-mining and statistical advances in doing phase analysis that optimize both the runtime and accuracy of SimPoint as well as target the overall simulation time. We present an approach that finds a single set of simulation points to be used across all binaries for a single program. This allows for simulation of the same parts of program execution despite changes in the binary due to ISA changes or compiler optimizations. Finally, we present a method of characterizing the behavior of parallel applications and use it to pick simulation points to guide multi-threaded simulations.

# I

# Introduction

Understanding the cycle level behavior of a processor during the execution of an application is crucial to modern computer architecture research. To gain this understanding, researchers typically employ detailed simulators that model each and every cycle of the underlying machine. Unfortunately, this level of detail comes at the cost of speed. Even on the fastest simulators, modeling the full execution of a single benchmark can take weeks or months to complete, and nearly all industry standard benchmarks require the execution of a *suite* of programs. For example, the SPEC CPU 2000 benchmark suite consists of 26 different programs, requiring the execution of a combined total of approximately 6 trillion instructions. Still worse, architecture researchers need to simulate each benchmark over a variety of different architectural configurations and design options, to find the set of features that provides an appropriate trade-off between performance, complexity, area, and power. The same program binary, with the exact same input, may be run hundreds or thousands of times to examine how, for example, the effectiveness of a given architecture changes with its cache size. To appreciate the significance of this problem we need a basic understanding of the benchmarks and the simulator used to model the processor architecture and estimate its performance.

## I.A Motivation

Processor architecture research quantifies the effectiveness of a design by executing a program on a software model of the architecture design called an architecture simulator. It is difficult to accurately compare studies that provide results for different sets of programs. To set a standard in the community, the Standard Performance Evaluation Corporation (SPEC) was established to provide a collection of benchmarks to evaluate processor performance. In the same manner, the architecture simulator needs to have a common baseline. SMT-Sim [70] and SimpleScalar [7] are two common cycle level processor simulators that have become standard models for architecture research. In this dissertation we will focus on the SimpleScalar simulator as it was used for the majority of experiments.

### I.A.1 SPEC CPU Benchmarks

The SPEC CPU 2000 benchmark suite has 26 programs, of which 12 are integer programs (primary execution is of integer instructions) and 14 are floating-point programs (primary execution is of floating-point instructions). The benchmark suite is chosen to stress a processor across its many components in a rigorous manner. Each program in the suite has 3 different inputs: *test*, *train*, and *reference*, which respectively correspond to a short test, a more representative training, and a full reference run. The test, train and reference inputs typically execute on the order of a few million, a few billion, and hundreds of billions of instructions respectively. Tables I.1 and I.A.1 show all the SPEC CPU2000 benchmarks, divided into integer and floating-point programs. The tables provide a high level description of each benchmark, its source language, and the number of instructions executed (in billions) with the reference and test inputs. These programs were compiled for the Alpha Instruction Set Architecture (ISA) with full optimizations. On average, the reference inputs execute for 223 billion instructions. The program `parser` has the maximum instruction count at 546 billion instructions.

Table I.1: SPEC CPU2000 Integer Benchmarks (lengths in billions of instructions)

| Benchmark | Ref Length | Test Length | Language | Category |
|---|---|---|---|---|
| bzip2 | 143 | 8.82 | C | Compression |
| crafty | 191 | 4.26 | C | Game Playing: Chess |
| eon | 80 | 0.09 | C++ | Computer Visualization |
| gap | 269 | 1.17 | C | Group Theory, Interpreter |
| gcc | 46 | 2.02 | C | C Programming Language Compiler |
| gzip | 84 | 3.37 | C | Compression |
| mcf | 61 | 0.26 | C | Combinatorial Optimization |
| parser | 546 | 4.20 | C | Word Processing |
| perlbmk | 111 | 2.0 | C | PERL Programming Language |
| twolf | 346 | 0.26 | C | Place and Route Simulator |
| vortex | 118 | 9.81 | C | Object-oriented Database |
| vpr | 84 | 0.69 | C | FPGA Circuit placement and routing |

Table I.2: SPEC CPU2000 Floating-Point Benchmarks (lengths in billions of instructions)

| Benchmark | Ref Length | Test Length | Language | Category |
|---|---|---|---|---|
| ammp | 326 | 5.49 | C | Computational Chemistry |
| applu | 223 | 0.18 | Fortran 77 | Parabolic / Elliptic Partial Differential Equations |
| apsi | 347 | 5.28 | Fortran 77 | Meteorology: Pollutant Distribution |
| art | 41 | 1.48 | C | Image Recognition / Neural Networks |
| equake | 131 | 1.44 | C | Seismic Wave Propagation Simulation |
| facerec | 211 | 4.12 | Fortran 90 | Image Processing: Face Recognition |
| fma3d | 268 | 0.00 | Fortran 90 | Finite-element Crash Simulation |
| galgel | 409 | 4.34 | Fortran 90 | Computational Fluid Dynamics |
| lucas | 142 | 3.71 | Fortran 90 | Number Theory / Primality Testing |
| mesa | 281 | 2.88 | C | 3-D Graphics Library |
| mgrid | 419 | 16.77 | Fortran 77 | Multi-grid Solver: 3D Potential Field |
| sixtrack | 470 | 8.59 | Fortran 77 | High Energy Nuclear Physics Accelerator Design |
| swim | 225 | 0.43 | Fortran 77 | Shallow Water Modeling |
| wupwise | 349 | 3.63 | Fortran 77 | Physics / Quantum Chromodynamics |

### I.A.2 SimpleScalar

SimpleScalar is a program that models the cycle level execution of a processor. It takes as input a program-input pair and simulates the execution from beginning to end, while computing relevant statistics for architecture research, such as cycles per instruction (CPI), cache miss rates, branch mispredictions, and power consumption. SimpleScalar has several models to represent different levels of execution detail. At the coarsest level of detail, *sim-fast* models only the functional execution of a program at the instruction level. A more detailed level, *sim-cache*, models the memory hierarchy and computes miss rates for those structures. The level of highest detail, *sim-outorder*, models the cycle-level out-of-order execution of a super-scalar processor. It is a superset of all the other models and provides the highest level of execution detail. The architecture research community uses SimpleScalar extensively, and today it is considered a standard architecture simulator.

The different models in SimpleScalar each have a stable execution rate. The fastest model, *sim-fast*, executes on the order of tens of billion instructions per hour on a 1 GHz machine. The slowest yet most accurate model, *sim-outorder*, executes on the order of hundreds of million instructions per hour, which is several orders of magnitude slower than the native hardware. It would take months of computation time to simulate the entire SPEC benchmark suite with *sim-outorder*. What makes matters worse is that researchers need to evaluate many different hardware configurations to measure the effectiveness of a design. This enormous turnaround time for a study makes simulating the full benchmark infeasible, and the majority of researchers only simulate a few hundred million instructions from each benchmark.

### I.A.3 Future Trends

SPEC periodically releases a benchmark suite to evaluate current and future processors. The SPEC CPU benchmarks have steadily increased in execution length, proportionally to the rate of processor speed increase. Given the SPEC measurement tools, small changes or fluctuations in the measurements have significant impacts on the percentage of improvements being seen.[1] Therefore, SPEC designs a benchmark suite that executes the reference input long enough on native hardware to achieve a valid timing.

Figure I.1 shows a projection of processor rate (in MHZ) and the SPEC benchmarks average execution length (in billions of instructions) in the years to come. The y-axis is logarithmic, and the x-axis shows time in SPEC suites that have been released (SPEC 92, 95, 2000, and 2006). Based on the data that has already been released, projections are made for the next 5 years with best-fit curves. It is expected that the rate of increase for the metrics will be exponential.

The SPEC CPU 95 suite instruction count stands out from the trend of the other SPEC suites. This benchmark suite executes on average 9.5 billion instructions and had programs that execute for only a few seconds on native hardware during the years it was active. This is too short an execution time to achieve valid timing measurements as mentioned above. Consequently the following suite, SPEC CPU 2000, had a substantial increase in instruction count to an average of 223 billion per benchmark. The recently released SPEC CPU 2006 suite executes on average over a trillion instructions and the longest running benchmark executes over 4 trillion instructions. We can speculate for the next generation SPEC CPU suite to execute many trillions of instructions per benchmark. Processors will also scale in speed, which means simulators will execute faster. But, it is also likely that simulators will scale in complexity which will

[1] http://www.spec.org/cpu2000/press/faq.html

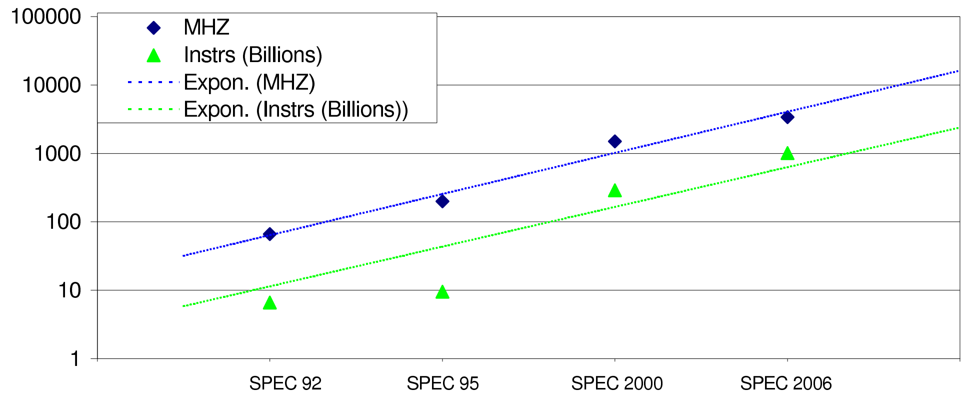


Figure I.1: Projection of processor speed and average instruction count for SPEC benchmark suites.

counteract the processor speedup. This is strong motivation for the continuing development of accurate, efficient, and robust simulation methodologies.

## I.B SimPoint

Researchers need techniques which can reduce the number of machine-months required to estimate the impact of an architectural modification without introducing an unacceptable amount of error or excessive simulator complexity. Until recently researchers would simulate SPEC programs for an arbitrary number of instructions (e.g. 300 million instructions) from the start of execution, or fast forward 1 billion instructions to try to get past the initialization part of the program. These techniques can result in error rates of up to 3736% in predicting the architecture metrics we wish to measure. We developed a tool, distributed as a software package called SimPoint, that enables both accurate and efficient processor simulation. SimPoint achieves this by exploiting the structured way in which individual programs change behavior over time.

As a program executes its behavior changes. These changes are not random, but rather are often structured as sequences of a small number of recurring behaviors, which we term *phases.* Identifying this repetitive and structured behavior can be of great benefit, since it means we only need to sample each unique behavior once to create a complete representation of the program's execution. This is the underlying philosophy of SimPoint [60, 61, 52, 23, 36, 34, 55, 53]. SimPoint intelligently chooses a very small set of samples from an executed program called *simulation points* that, when simulated and weighted appropriately, provide an accurate picture of the complete execution of the program. Simulating in detail only these carefully chosen simulation points can save hours of simulation time over a random sampling of the program, while still providing the accuracy needed to make reliable decisions based on the outcome of the cycle level simulation. SimPoint achieves very low error rates (within 2% average error for the SPEC benchmarks) and on average reduces simulation time by a factor of 1,500, compared to simply simulating the whole program. This approach is now used by researchers in the architecture community, and companies such as Intel [48].

## I.C Thesis Overview

This dissertation presents how repetitive phase behavior can be found in programs through machine learning and describes how SimPoint automatically finds these phases and picks simulation points. This method forms a foundation that we extend to meet important needs in architecture research.

We describe data-mining and statistical advances in doing phase analysis that optimize both the runtime and accuracy of SimPoint as well as target the overall simulation time. We present an approach that uses statistical analysis and sampling to guide choosing the number of clusters and to provide a confidence and probabilistic error bound for a given clustering. We also present an approach that finds a single set of simulation points to be used across all binaries for a single program. This allows for simulation of the same parts of program execution despite changes in the binary due to ISA changes or compiler optimizations. Finally, we present a method of characterizing the behavior of parallel applications and use it to pick simulation points to guide multi-threaded simulations.

The remainder of this thesis is laid out as follows. We begin with a background of efficient simulation techniques and discuss their effectiveness in Chapter II. Chapter III defines time varying program behavior and presents the base approach for characterizing this behavior and finding simulation points. Chapter IV describes statistical and data mining advances that both optimize the algorithm, calibrate simulation accuracy, and reduce overall simulation time. Chapter V extends the approach to handle variable length intervals which are used to find a single set of simulation points across multiple binary versions of a program. Chapter VI describes how we characterize the execution of parallel applications and find simulation points across the parallel threads. We summarize this thesis in Chapter VII and explore future directions.

# II

# Efficient Simulation Background

The combination of long benchmarks, large design space search, and slow simulation has motivated research in the area of efficient processor simulation. Until recently it has been accepted to simulate only an arbitrary portion of a benchmark. For example, the first billion instructions are simulated in detail to represent the entire run. This technique does a poor job in representing the entire execution since programs generally have an initialization phase at the start of the execution. Depending how long the initialization phase is, the first billion instruction will generally over-represent the initialization phase rather than the steady state of program execution. A less guaranteed improvement is to skip an arbitrary number of instructions (e.g. 1 billion) to forgo the initialization period, and then simulate in detail a billion instructions. In recent studies [60, 61] we have shown program execution behavior to vary over large regions (billions of instructions) and capturing the program execution for a billion instructions after initialization phase can represent only a small fraction in the overall program behavior. These techniques have been shown to miss-estimate performance by up to 80% on average.

This chapter surveys current and past techniques proposed to optimize processor simulation. These techniques can be categorized into three general types:

1. Reduced input: Reduced input size can reduce execution duration, since the benchmark inputs have a direct impact on the execution duration.

2. Statistical simulation: Generate a synthetic representation of a benchmark then simulate it on a statistical model of the processor.

3. Sampling: Simulate only samples of a benchmark.

Of the three simulation optimization types, sampling has received the most attention. The approaches taken to perform sampling vary widely, and can be further categorized into three techniques:

1. Statistical sampling: Well known sampling techniques (e.g. random sampling) applied to program execution.

2. Representative sampling: Reduce the sample population by first characterizing the behavior of execution, then sample from the different behaviors observed.

3. Sample startup techniques: Means to get to a sample and then warm up the state of the processor before detailed simulation. A sample may be deep in the execution and getting to it efficiently is critical for reducing simulation time. The *cold start* effect is the state in which a structure such as the cache or branch predictor start out when a program starts executing. These structures warm up during the execution with memory accesses and execution patterns unique to the program, and their performance improves. The state of these structures can have a significant impact on the overall performance (CPI).

The remainder of this chapter is outlined as follows. The reduced input techniques are described in Section II.A. Statistical simulation techniques are describe in Section II.B. Section II.C describes sampling techniques, including statistical, representative sampling, and sample startup techniques. A comparison of the various techniques is discussed in Section II.D. Conclusions are summarized in Section II.E.

## II.A Reduced Input

Each program in the SPEC 2000 CPU benchmark suite executes more than 200 billion instructions on average with the reference input. The programs execute only 3.5 million instructions on average with the test input. This disparity exposes the potential to reduce execution duration with modified inputs. The MinneSpec project [27, 31, 30] strives to reduce simulation time with specially designed inputs that closely represent program execution with the reference input but at a fraction of execution duration. The goal of the work is to have 3 inputs similar to the ones SPEC provides, but on a smaller simulation time scale: small input simulates for a few minutes, medium input simulates for a few hours, and large input for a few days. The small input is expected to be used as a quick test without much accuracy in representing the entire benchmark. The medium input is more representative than the small input. The large input is expected to closely represent the reference input.

Before generating the reduced input set, a set of profiles are first collected for each benchmark. These profiles are used to measure how representative a reduced input is. Each benchmark is compiled at four different optimization levels, O0 through O3. Each binary is then simulated with each of the three inputs (test, train, and reference) using sim-fast to measure instruction counts. The four different compilations of the binaries are simulated only with the reference input using sim-profile, a more detailed functional simulator, to collect instruction mix profiles. These include totals for different instruction classes (e.g. loads, stores, conditional/unconditional branches, and integer/floating-point computations) as well as individual counts for each unique instruction. Memory profiles are collected for a number of different cache configurations with sim-cache, a functional and memory hierarchy simulator. Finally, the percent of execution spent in each function is measured using *gprof*.

After the profiles are collected, the reduced inputs are manually generated. To generate reduced inputs, several strategies are employed depending on the benchmark. For benchmarks that have command line inputs, the command line parameters were varied to shorten execution. If the benchmark does not have a command line input then the input file can be reduced by sampling. In some cases an entire input file needs to be created from scratch.

The result of simulating reduced inputs is validated with the chi-squared test. This test compares the distribution of the reduced input profiles to the profiles collected for the reference input. The heuristic used for validating the 3 reduced inputs (small, medium, large) is that the inputs have progressively better chi-square test scores. The small input may have the largest acceptance bound, the medium input needs a reasonable score, and the large input must have a high score at 90% confidence level to accept it as a valid reference substitute.

Of the 32 benchmark configurations evaluated, only 18 of the large inputs satisfied this requirement. Several other metrics are used for evaluation, including the instruction mix and memory behavior (e.g. miss-rate) across different cache configurations. Most of the reduced input's instruction mix profiles match the reference profiles. However, the memory profiles reveal that the footprint of the reduced inputs is much smaller than the reference ones. This difference has a significant impact on the cache miss-rates and other memory behaviors. Simulations which are sensitive to memory behavior can produce different results with the reduced inputs than with the reference inputs. The authors suggest that the reduced input set be considered as a separate workload for such experiments, and not a representative of the SPEC benchmark.

## II.B Statistical Simulation

A different approach to reduce simulation time is to target the speed of detailed simulation. Statistical simulation achieves this by using a probabilistic model of the processor states. Statistical simulation is not as accurate as discrete simulation (e.g. SimpleScalar), but is suitable for exploring a large design space. Due to its probabilistic nature, statistical simulation runs synthetic benchmarks that are also based on probabilistic models. A synthetic benchmark statistically represents a real benchmark, based on profiles collected for the real benchmark using detailed simulation. Various metrics are profiled for benchmark reference-input pairs, then statistically analyzed and modeled in the synthetic benchmark. The resulting synthetic benchmark generates similar statistics for each of the metrics targeted. This section describes several techniques proposed in synthetic benchmark generation and probabilistic processor models.

### II.B.1 Probabilistic Simulation

Noonburg and Shen [45] propose a framework for statistical simulation of processor performance. Their statistical model uses processor states similar to those in trace-driven simulators. Instead of discrete time based states, they compute probabilities for each state. The cycles spent in different states can be computed from these probabilities.

The first step in this framework is to model the processor with processor components. A connection is defined as several components (e.g. issue buffers, pipelines, etc.) that are modeled as blocks, and then connected with edges. For example a connection can represent an issue buffer component connected to several pipeline stages of execution. The component-connection forms the foundation of their framework which can be extended to model complex processor systems (much like lego pieces combined to make larger models). The number of instructions moving through a connection at a given time is measured with the *flow*. The flow is based on the number of instructions that are ready to be issued into the connection, the number of instruction that may be blocked by dependences or structural hazards, and the bandwidth of the connection. The flow is computed as the minimum of the three parameters. To model the processor, a Markov model is composed using the processor states and probabilities of transitions between pairs of states. The transition probabilities between possible states is computed from a program trace where instruction type and dependency distance from previous instructions are profiled. This model forms a probabilistic representation of the processor model.

Estimating performance on the Markov processor model is computed with the flow and state-transition probabilities. The flow probability distribution is computed from the probability of the processor being in a given state as well as the transition probabilities between states. This is similar to the probability of a number of instructions flowing from one component to another. The CPI is computed as the weighted average of the flow probability distribution. Experimentally this model performs with reasonable accuracy for the simple processor models tested. However, there are two main issues with the model. First, the space required to store the transition probabilities for the Markov model is $n^2$. Even though the transition probabilities are mostly sparse, this emerges as the bottleneck when the processor model becomes more complex and more states are introduced. Second, the accuracy of the model has not been verified on complex processor models, but on simplistic models it has been shown to arrive within 10% of the true performance.

### II.B.2 Symbolic Execution

A different statistical simulation technique, HLS [47], combines statistical profiles of applications with symbolic execution. At a high-level this approach models a baseline architecture similar to the one in SimpleScalar [7], but where several of the components in the processor are statistically modeled instead of discretely implemented. These include the level 1 data and instruction cache, the level 2 unified cache, main memory, and the branch predictor.

HLS creates a synthetic benchmark for each real benchmark. To generate the synthetic benchmark, profiles are collected of a real benchmark, the profiles are interpreted, and then a synthetic code sample is generated. The profile collection process uses SimpleScalar to gather basic block size and distribution as well as a histogram of the distances between dynamic instructions of different types (integer, floating point, load, store, and branch). A more detailed profile is collected to measure cache behavior and branch prediction accuracy. The distance between an instruction and the instructions that depends on it is a critical parameter in HLS. This parameter characterizes how much parallelism an application has, and is a major influence on performance. The statistics computed from the profiles collected (e.g. average size of a basic block with standard deviation) are used to generate the synthetic trace of the application.

Once statistical profiles are generated, the synthetic trace can be constructed. The synthetic trace has instructions contained in basic blocks that are linked together into a static program control-flow graph. Instead of having specific arguments to be operated on, the instructions contain a set of statistical parameters. A function unit requirement parameter specifies what functional unit in the core is required to execute the instruction. This parameter is based on histograms of the different instruction types profiled. The dynamic instruction distance parameter defines the distance between the current instruction and the previous instruction which it depends on. A constraint is placed on this parameter so that any instruction will not have a dependency on a store or a branch, since these dependencies do not exist in real programs. This parameter quantifies how many instructions back a data dependency is (e.g. register read-after-write), and does not provide any more detail as to what kind of dependency it is. Finally the size of each basic block (and standard deviation) is computed from the profiles collected. Each basic block terminates with a branch, which has branch predictability assigned to it based on the branch prediction accuracy of the application. The memory is modeled as a normal distribution for each cache, with an average hit rate equal to that of the simulated hit rate.

The synthetic trace once generated can be executed on the processor model. As mentioned earlier, the architecture closely resembles the baseline architecture in SimpleScalar. The model supports out-of-order execution, and has five major stages: fetch, dispatch, schedule, execute, and complete. The instructions in the synthetic trace execute in the model as a conventional instruction would. Instructions are fetched and sent to dispatch stage. The dispatch interprets the instructions and then they are sent to reservations stations until all their dependencies are resolved. Instructions then proceed to their parameter specified functional unit to be executed. Finally, their result is sent to the completion unit if bus bandwidth is available.

The performance of the HLS model quickly converges during execution. Experimentally it is shown that after one thousand cycles the IPC is nearly converged, and after six thousand cycles it has become constant. The accuracy of HLS is estimated by comparing its performance estimations to detailed simulation performance estimations using SimpleScalar. The average error in IPC for 8 integer SPEC 95 programs is 2.75% and 4.76% for the test and reference inputs respectively. It is noted by the authors that performance accuracy is correlated to specific hardware component performance (branch prediction accuracy, level 1 data and instruction cache hit rates). The more accurate these components perform, the higher the accuracy of HLS is in modeling the overall performance (IPC). It is also noted that while the processor modeled in HLS closely resembles SimpleScalar, it can be configured to model other processors. The MIPS R10K processor was modeled, and performance estimations were within 3.2% on average to the native hardware performance with the SPEC CPU 95 reference input.

### II.B.3 Statistical Simulation Optimizations

In Nussbaum and Smith [46] analyze statistical simulation models to expose sources of error and address them. They propose a refined statistical model which achieves higher accuracy in modeling processor performance.

Initially a baseline statistical model is implemented using a simple instruction mix and interdependency profiles to generate a synthetic trace. There are 14 instruction types (e.g. nop, integer-alu, integer-multiply, integer-divide, fp-add/sub, fp-multiply, branch, system call) and their interdependencies are probabilistically projected into the future. Each instruction has some probability associated with it for a future instruction being dependent on it, up to the capacity of the instruction window in instructions succeeding it. Instructions further in the future than the size of the instruction window will not have any dependencies exposed during execution. The processor model is a simplified version of Simple Scalar [7]. Hardware specific characteristics (e.g. cache and branch prediction behaviors) are input parameters to the statistical simulator, since these structures are architecture dependent and are not incorporated into the synthetic benchmark (unlike instruction characteristics that are architecture independent such as instruction mix).

The baseline statistical simulation model is analyzed by checking accuracy of each statistical component independently and then attributing its effect on the overall accuracy. To this end, a detailed synthetic simulation is run where instead of executing a synthetic trace, a real trace is mapped to the simplified instruction types and simulated cache and branch predictor characteristics are incorporated to each instruction in the trace. This detailed synthetic simulation is the upper bound in accuracy a synthetic trace can achieve. Then statistical modeling was introduced one step at a time. For example, the detailed synthetic simulation was modified with a statistically modeled cache or statistical branch prediction or statistical instruction dependencies. This is done independently with each statistical model and also as a mix of all three. Experimentally these simulations are compared to a detailed simulation with SimpleScalar, and the results show that the three statistical models produce different amounts of error depending on the program. Additionally, there is no reliability in these errors (e.g. statistical cache does not always produces more error than statistical branch prediction). The trace containing all three statistical liberties (the mixed trace) is the target of the evaluation, since it is the most efficient synthetic trace to simulate. The error for the mixed trace varies widely across the Spec 95 benchmarks, where a few programs (e.g. gcc, compress, and tomcatv) have more than 10% error in IPC. These errors are attributed to particular behaviors in the programs, for example `compress` spends a billion instructions in the initialization phase with an unbalanced ratio of branch mispredictions. Statistically, the mixed trace targets an agglomerated average of the entire execution behavior, and it substantially differs from the real execution behavior.

To address the sources of error in the baseline statistical simulation, a refined model is designed targeting specific sources of error. One such refinement addresses the size of basic blocks in the trace. Simplistic models will consider only the mean and standard deviation of basic block sizes. However, basic block sizes often vary across an execution, and are not normally distributed. To improve the basic block size distribution in a synthetic trace, a set of distributions are collected based on the history of recent branch outcomes. This ingrains spatial locality into the distributions, since branch histories will be similar when they are executing similar regions in the code space. Another refinement improves the instruction mix distribution. Real applications do not have a random instruction mix throughout execution, but instruction mix that are structured and dependent on the code region. To improve the instruction mix distribution in the trace, the probability of each instruction type is made a function of the distance from the immediately preceding branch. Another refinement is to keep different instruction mix distributions for each basic block size. To improve instruction dependency modeling, similar methods can be taken; make the dependency distribution a function of basic block size. Similar approaches can also be taken with the cache and branch prediction modeling. The authors conclude that the basic block size distribution based on the recent branch history has the greatest improvement in accuracy. Simplistic models that simulate fast can forgo this refinement as a trade off for speed. To achieve a model with greater accuracy all the refinements should be employed at the cost of greater simulation time. The increased simulation time comes from the extra time it takes for the performance metrics to converge with the non-normal distribution models.

Eeckhout et al. [16] further optimize the statistical simulation by modeling the control flow. In their approach a statistical flow graph is generated from basic block history profiles. The graph has a node for each basic block, and edges between nodes are weighted by the probability that a block will transition to the other block. A node with edges coming out of it will represent all the possible transitions out of a basic block, and the sum of the transition probabilities equals one. This model is incorporated into the synthetic trace and provides a closer resemblance to real program behavior.

## II.C Sampling Techniques

Sampling is an established method for representing a data set with a fraction of the data. It is not surprising that sampling has been applied to lengthy architecture simulation. Sampling techniques are split into two general types, statistical sampling and representative sampling. Statistical sampling samples the execution in a random or systematic pattern without special consideration of what sample is picked. Representative sampling, on the other hand, carefully picks samples that are used to represent the execution. In either method it is essential to have good sample startup; efficient means to get to the sample and warmup the state of the processor to avoid cold start bias. The term sample is defined here as a contiguous interval of dynamic instructions during program execution.

### II.C.1 Sample Startup

There are two essential components for sampling techniques to be effective. Getting to the sample must be significantly faster than detailed simulation otherwise speedup gains will be dampened. Then before a sample is simulated in detail it is necessary to warmup the caches and branch predictors so that they perform at a similar level as if they were functioning up to before the sample start point.

#### Getting to the Sample

Sampling techniques are effective by simulating only a small fraction of the entire execution. If a sample is deep in the execution, it is necessary to have the means to reach it for detailed simulation. It is critical that substantial time is not invested in reaching the sample, otherwise that time will reduce the speedup gain from sampling. Here several methods for efficiently getting to the start of the sample are described.

Cycle level simulation takes on the order of a thousand times slower than native hardware. On the other hand, a functional emulator, such as sim-fast, only emulates instructions and is substantially faster (a couple order of magnitude) than the cycle level simulator. When emulating the program it is referred to as *fast-forwarding* through execution. Fast-forwarding is one method that can be used to reach the start of a sample before simulating it in detail. Some benchmarks have hundreds of billions of instructions and if a sample is deep in execution it can take days to fast forward to it.

An attractive alternative to fast-forwarding is checkpointing. Checkpoints are a snapshot of the architecture state at a given point in the execution- the pipeline, buffers, and structures like the branch predictor are stored in a file. To generate a checkpoint one first has to get to that point in the execution (e.g. via fast-forwarding) and then flush the checkpoint. Checkpointing is an attractive method because once it is created it can be loaded and simulation starts from the point of the checkpoint without significant time overhead. This can also be done in parallel resulting in simulation of multiple samples in parallel. It is a speedy alternative to fast-forwarding, with the trade off of size.

A naive approach to checkpointing requires substantial space since it stores the entire state of the processor. Several techniques have been proposed to minimize this overhead in [71]. The *Touched Memory Image* method only stores the blocks of memory and their addresses that are to be accessed during a sample that will be simulated. This technique reduces the storage space required since it uses a sparse image of the entire architecture state that will be sufficient for the sample to start execution in a warm state. A further refinement in this method is to only store memory blocks that are written to after being read, since in the read after write case the original data in that memory block will not be utilized. A second approach, called the *Load Value Sequence*, stores a log of all data values loaded from memory during an execution. During execution this log can be replayed concurrently and provide the loads with the correct data values. Further optimization on this technique attaches a bit for each load to decide whether or not the data will be read from the log. This method cuts out subsequent loads and stores accessing the same memory address or when the loaded value is zero (the initial memory value in a simulator). Both of these techniques reduce the storage required for the full checkpoint by two orders of magnitude.

Several techniques propose to use the native hardware to speed up fast-forwarding. SimSnap [68] is a technique that leverages the native hardware to execute between samples of detailed simulation instead of fast-forwarding. This method transitions from native to detailed simulation and back via checkpoints. The binary is slightly modified to dump checkpoints at sample entry points with a specialized compiler. In [20] a direct execution model is used to model processor behavior and performance directly from the native hardware. In this work a program source is translated to an intermediate representation (e.g. assembly) and then instrumented with timing code to collect timing statistics during execution. The timings statistics are translated to the modeled hardware to estimate its performance. This is an involved process since the mapping of native-to-model timing is required for every instruction type that has different execution time on the native and modeled processor. Although such a technique is more efficient than simulation, it has several handicaps: the modeled processor is highly limited by the native processor design. Both SimSnap and the direct execution model are limited to the same ISA in the simulation model and native hardware.

One method that is not restricted to the same ISA while exploiting hardware in simulation is Embra [73]. This technique uses dynamic binary translation (JIT) to generate code sequences to simulate a benchmark. Instead of interpreting the benchmark's instructions as a simulator would, the binary translation converts the code executed on the native hardware to simulate the execution of the original instructions. This process is fast, but it suffers from a high level of complexity when doing the binary translations.

### II.C.2 Sample Warm Up

The cache and branch predictor in a processor are structures that require training before achieving peak performance. When a program starts execution all the data accesses it performs are likely to miss in the cache the first time they are referenced. Similarly, the branch predictor is not going to have any branch history to make an educated prediction the first time each branch is encountered. After reasonable execution time, both the cache and branch predictor have encountered sufficient events to make them effective. If a sample begins detailed simulation in a cold start state (e.g. the caches do not not have any recently accessed data already loaded or the branch predictor is not trained with the execution control-flow patterns), it may under represent performance. Several *ad hoc* techniques have been employed to address this effect: ignore cold start and assume a cold state at the start of each sample, which should not impact large enough samples; preserve the state between samples, the next sample starts with the state at the end of the last sample; consider every first cold access in the cache or branch prediction as a hit (e.g. the first time a memory address is accessed consider it to be already loaded in the cache, or the first time the branch prediction is made for a branch it is correct); perform function simulation for a constant instruction window size (e.g. 1 million instructions) before each sample. Function simulation is fast because it only models the structures (e.g. cache and branch predictor) and does not model the cycle detailed behavior or the pipeline.

In the first technique using a long interval length (e.g., more than 100 million instructions) will make warmup unnecessary for many programs. This is the approach used by Intel's PinPoint for simulation [48]. They simulate intervals of length 300-500 million instructions so they do not have to worry about implementing warmup in their simulation infrastructure. With such long intervals the architecture structures are warmed up sufficiently during the beginning of the interval's execution to provide accurate simulation results. In the latter technique it is likely that either too many instructions are being used for warmup and simulation time is being unnecessarily increased, or that the structures have not warmed up within the given warmup stage and the sample statistics may not be accurate. The work done by Haskins and Skadron [24, 25] addresses these issues and propose a technique to accurately gage how long to warmup to achieve high accuracy.

In [24] a warmup acceleration technique, Minimal Subset Evaluation, is proposed to achieve an accurate cache state by computing the number of references prior to a sample that must be modeled before detailed simulation. This technique does not target the secondary cache (unified or split) nor the branch predictor. A more recent technique, Memory Reference Reuse Latency [25], applies similar methods that target all caches and the branch predictor. This technique measures memory reference reuse latencies, the number of completed instructions between consecutive references to each unique memory location, to determine at what point prior to the sample warmup should start. This data is collected for the instruction stream (instruction-cache), data stream (data-cache), and branch stream (branch-predictor). Statistically analyzing the data provides probabilistic models that X% (e.g. X=99%) of the addresses will be accessed within an instruction window of size N. The proposed method to use this warmup period is to reach the point in the execution that is N instructions before the start of the sample (e.g. via fast-forwarding), then run a function-level detail simulation to warm up the caches and branch predictor for N instructions. At this point, the start of the sample is reached and detailed simulation can begin with warmed up structures. The authors note that for large X%, (where $X = 99.9\%$) , a disproportionate amount of instructions would need to be warmed up to compensate for the remaining 0.1% addresses and ignoring these addresses has insignificant impact on the sample warmup state. This technique reduces simulation time significantly by achieving 90% of the maximum potential speedup had no warmup period been used and also accurately models the full detailed simulation state at the end of the warmup period to within 1%.

### II.C.3 Statistical Sampling

Statistical sampling established long before processor architecture research, and benefits from a rigorous mathematical foundation. Conte [11] pioneered the use of statistical sampling in processor simulation. In that work sampling methods were applied to cache and instruction traces with good accuracy. The drawback in that approach is that it required a full trace *a priori* for the sampling analysis. That means that a full trace of the metric considered (e.g. cache missrates or IPC) needs to be collected with detailed simulation to completion.

In more recent work, Conte et al. [12] provided a framework to employ statistical sampling on the fly, thus absolving the need for full trace generation. First, random samples from the execution are simulated in detail where each random sample is a contiguous region of dynamic instructions. Statistical metrics are computed for the samples to predict accuracy of estimated results. The metric of interest (e.g. IPC) is statistically analyzed. Standard deviation and probabilistic error and confidence bounds are computed to estimate the accuracy of the sample set. At a high level, the more samples collected the more accurate the estimate will be. This is based in part on the *Central Limit Theorem* stating that the sum of variables with a finite variance will have an approximate normal distribution. We will describe the confidence and statistical error bounds in more depth in Chapter IV.

The authors specify two sources of error in their sampling technique: non-sampling bias and sampling bias. Non-sampling bias results from the cold-start effect. This issue was already discussed in Section II.C.2. Sampling bias on the other hand is fundamental to the samples, since it quantifies how accurate the sample average represents the overall average. There are two major parameters that influence this error: the number of samples and size of each sample in instructions. The size of a sample in processor simulation is the number of dynamic instructions that a sample encompasses. The authors experimentally determine these parameters for their processor model and conclude that 1,000 samples of 2,000 instructions is sufficient for accurately representing a benchmark. They also provide a general method to determine the sample count on other models. This method is based on a heuristic, in which the user determines a particular accuracy level with which the metric of interest will be estimated. The benchmark is then simulated, and N samples are collected. N is an initial value for the number of samples. Error and confidence bound are computed for the samples, and if they satisfy the accuracy limit then this estimate is good. Otherwise, more samples need to be collected ( $>$ N) and the error and confidence bounds are recomputed for each sample set collected until the accuracy threshold is satisfied.

An automated approach for applying a similar sampling technique is proposed in the SMARTS [74] framework. This work uses systematic sampling and reduces sample startup overhead by maintaining certain structures (cache and branch predictor) warm throughout the execution. This minimizes the need to warmup with detailed simulation prior to sample startup. The authors experimentally determine a smaller sample size of 1,000 instructions. The smaller sample is due in part to the more effective warming of the processor between samples. Detailed warming is still essential for the samples. This period is affected by the processor architecture. Depending on the issue width of the processor, it is recommended that an 8-way processor have 2,000 instructions for detailed warming and a 16-way processor have 4,000 instructions prior to each sample. A similar method as in [12] is proposed to determine the number of samples needed to satisfy a particular accuracy level. The authors recommend a 10,000 sample count to achieve a $\pm 3\%$ error bound at 99.7% confidence level on the SPEC CPU benchmarks.

### II.C.4 Representative Sampling

Representative sampling contrasts statistical sampling in that it first analyzes the data in order to pick a few but highly representative samples. The key advantage in this approach is that having fewer samples can reduce simulation time and also allow for a simpler simulation infrastructure. The number of instructions simulated in detail is not the target here, but rather the number of samples needed for simulation and the total overhead in sample warm up. For example, for ten samples it is fairly fast and easy to either fast-forward or collect checkpoints and then simulate them in parallel (in contrast to thousands of small samples that may be needed with statistical sampling). An instruction interval in this section is defined to represent a contiguous stream of instructions during program execution.

The SimPoint method [60, 61, 52, 6, 36, 34, 55, 53] is a representative sampling approach that automatically characterizes the behavior of a program and then samples each unique behavior once. This method is the focus of this thesis and we will describe related work for doing representative sampling in this section.

Lafage and Seznec [32] proposed a method for doing representative sampling that uses data stream profiles to pick a handful of samples. In their approach, metrics to represent temporal and spatial locality of data memory accesses is profiled across the entire program execution. For example, to measure temporal locality in data accesses the number of instructions between two accesses for the same address are measured for each address in the program. To collect the metrics it is necessary to model the memory hierarchy by simulating the memory structures (e.g. caches) and additional state to capture multi-configuration behavior. This can be done in a single pass through the simulator, but still imposes a significant overhead. During simulation, statistics are output at intervals of one million instructions. Each one million instruction interval can be represented as a vector with the statistics applied to dimensions in the vector.

In vector format, it is possible to compare similarity between vectors by computing the distance between the vectors. A small distance between two vectors means they are quite similar to each other and a large distance means the two vectors differ substantially. With the ability to measure similarity between vectors, a hierarchical clustering algorithm is applied to classify all the intervals into a set of clusters. Each cluster will have intervals in it that are similar to each other, but different from intervals in other clusters. The hierarchical clustering algorithm starts at the top level in the hierarchy where all the intervals are in a single cluster. Each progressive level in the hierarchy splits a cluster into two clusters of equal or greater similarity. Once the clustering hierarchy is complete, a sample is picked from each cluster as a representative for the cluster. The cluster representative is the vector that is closest to the center of the cluster. To pick a level in the hierarchy a metric measures how close the weighted mean of the representative samples is to the centroid of the cluster at the top of the hierarchy (all the vectors in one cluster). A weighted mean is used since the samples represent clusters of varying size. Each representative sample is weighted by the size of the cluster it represents. Finally, the set of representative samples is used to represent the entire execution.

One critical issue in this approach is the use of data stream profiles, an architecture dependent metric, for choosing representative samples. The representative samples picked based on these profiles are dependent on the architecture configuration that generated the profiles. If a design space exploration spans a wide spectrum of memory configurations (potentially hundreds or thousands of configurations), profiles would have to be collected for many different architecture configurations. Scalability of this approach can be highly limited in such scenarios.

In contrast, the SimPoint technique uses an an architecture independent metric, code profiles, to characterize program behavior and choose representative samples. This results in representative samples for the benchmark that are valid across many different architecture configurations. Additionally, architecture independent code profiles can be collected efficiently (e.g. with sim-fast), and take a fraction of the time that functional simulation would require. This method is described in detail in the next chapter.

## II.D Discussion

In this section a comparison between the various techniques is discussed as well as their effectiveness with future projections of SPEC benchmarks.

### II.D.1 Technique Comparison

Table II.D.1 shows a ranking of the various techniques under the following categories:

- **Set up overhead:** set up overhead such as preprocessing (e.g. collecting profiles) needed by the technique
- **Implementation:** overhead of designing and implementing the infrastructure essential for the simulation technique
- **Ease of use:** ease with which a simulation can be run using the technique
- **Time to simulate:** time it takes to complete a simulation as defined by the technique
- **Accuracy:** accuracy of technique in representing the full detailed simulation of the reference input of the benchmark

The following sections compares the different techniques under each of the categories, and analyze the rankings in table II.D.1.

#### Set Up Overhead

Set up cost for a technique includes the overhead of collecting profiles to represent a benchmark or the investigation required before implementation of the technique. The reduced input technique carries the highest set up costs. This technique requires extensive understanding of the benchmark to effectively reduce the execution duration with a modified input that is still representative of the reference input. This is a manual process that investigates the source code of the benchmark, and then validates the reduced input with profiles collected for the reference run. Other techniques, such as statistical simulation and representative sampling require profiles collected for the complete run and then analysis of the profiles. These techniques have a smaller set up cost since the profile collection is automated, and the time it takes depends on the detail of the profile. The profiles needed for the HLS [47] statistical simulation technique is more detailed than the profiles collected with the representative sampling technique proposed in SimPoint [60, 61, 52, 23], because they are used to generate a synthetic benchmark. Finally, the lowest set up cost is with statistical sampling, which can completely avoid set up cost by collecting profiles on the fly and decide post-simulation if the accuracy is sufficient.

#### Implementation

Implementation costs for simulation is a serious consideration since the overhead in implementing the technique may outweigh the time savings the technique provides. Statistical simulation is the most difficult technique to implement. It requires an overhaul of a detailed simulator to incorporate probabilistic models for certain structures. This overhead is similar in magnitude to the set up cost required in reduced input technique. Statistical and representative sampling techniques have an implementation overhead that is described in sample startup (Section II.C). Representative sampling has lower implementation costs since the overhead of warming up samples is avoided by using larger samples that are not vulnerable to the cold start effect. Reduced inputs have no implementation costs, since it simulates a benchmark-input pair to completion which requires no modification to the simulator.

#### Ease of Use

Ease of use measures how easy it is to perform the simulation once the infrastructure has been designed. Reduced input technique is the easiest technique to use. That technique requires no modifications to the simulator, and has no overhead in pre or post processing the simulation statistics. On the other extreme the most difficult simulation technique presented is statistical simulation. Statistical simulation requires two major components: a simulation infrastructure that probabilistically models the processor and a synthetic benchmark generated from profiles collected with detailed simulation. Once the processor model is implemented, each benchmark needs to be synthetically generated. This overhead outweighs the usage overhead in the other techniques. Sampling techniques need methods to get to every sample. Checkpointing or fast-forwarding would work for getting to samples as mentioned in Section II.C. Note that representative sampling uses orders of magnitude fewer samples than statistical sampling (tens vs. thousands), which in terms of checkpointing would require substantial less space. Finally, statistical sampling requires post-processing to determine if a set of samples satisfies a user-specified accuracy level.

**Time to Simulate**

What statistical simulation lacks in ease of use and implementation it makes up in efficiency. It is the fastest technique since performance converges after only thousands of instructions (a few minutes at most). This is orders of magnitude faster than the other techniques. The slowest technique is statistical sampling since it requires to go through the entire execution of the reference run of a benchmark. Fast-forwarding through some of the benchmarks can take days, and this outweighs the time spent in detailed simulation. If direct execution is implemented with statistical sampling, instead of fast-forwarding between sample the execution is driven on native hardware, then this technique would be much faster. Representative sampling is faster to simulate than statistical sampling, because with fewer samples there is less likelihood to have to simulate until the end of execution. Simulation time can be further reduced in representative sampling with checkpoints for each representative interval. It would require substantial space to have checkpoints for statistical sampling, since there are thousands of samples. Finally, the time to simulate the reduced input is independent from the duration of the reference run. Depending on the input (small, medium, or large), it can take from a few minutes to a few days to complete a simulation.

### Accuracy

The most accurate simulation technique is statistical sampling. It can achieve the best representation of the reference run, since it can increase the number of samples to match any level of accuracy (at the price of higher simulation time). Representative sampling is a close second in accuracy with better efficiency since only a handful of samples need to be simulated. Optimizations in representative sampling, as proposed in [52], which incorporate probabilistic confidence and error bounds to the SimPoint technique can match the accuracy of statistical sampling. Statistical simulation is a distant third, since it uses both a probabilistic processor model and a synthetic benchmark. A lot of optimizations have improved the accuracy in modeling the discrete simulation, but it still lacks the detail that sampling techniques have. Finally, reduced input is the least accurate technique, since the reduced inputs exhibit significantly different behavior from the reference input.

### Comparison Summary

The various simulation techniques presented have a wide range of characteristics where each has its own unique strength and weakness. Depending on the expected use for a simulation, each technique has its niche. For example, in doing a large design space exploration it may be worth the overhead to implement statistical analysis to quickly find Pareto Optimal points. If ease of use is an issue, then the reduced input technique can also be applied to exploring a design space, although it is not quite as efficient. Representative sampling with checkpointing will have comparable efficiency to reduced input, and can also be used for exploring a large design space. To accurately gage design decisions, statistical or representative sampling techniques are appropriate. These techniques have the best accuracy while significantly reducing simulation time from a complete detailed simulation.

Table II.1: Simulation optimization technique comparison

| Technique | Set Up | Implementation | Ease of Use | Time to Simulate | Accuracy |
|---|---|---|---|---|---|
| Reduced Input | 4 | 1 | 1 | 2 | 4 |
| Statistical Simulation | 3 | 4 | 4 | 1 | 3 |
| Statistical Sampling | 1 | 3 | 3 | 4 | 1 |
| Representative Sampling | 2 | 2 | 2 | 3 | 2 |

Representative sampling has become the method of choice in many cases because of its combination of strengths. It is both easy to use and implement and reduces simulation time to hours while achieving very good accuracy (2% error on average for SimPoint [23]). Intel [48] has embraced representative sampling for its simulation infrastructure because of these characteristics. It has also become a methodology standard in the architecture research community for the very same reasons.

### II.D.2 Effectiveness with Future Trends

Future releases of SPEC benchmarks will be longer and processor simulators will be more complex and slow as was projected in Section I.A.3. It is important that an efficient simulation technique is scalable to future benchmark releases. One technique which suffers from scalability is the reduced input technique (Section II.A). That technique is not automated, and has a high set up cost requiring extensive knowledge of each benchmark. That overhead will be carried over for each new benchmark that is introduced. Statistical simulation may also suffer from a high implementation cost, since more complex processor models may need to be probabilistically modeled. It is not as direct of a drawback though, since the complexity is dependent on the processor and not the benchmarks. Finally, the sampling techniques are automated enough to be efficiently applied to new benchmark suites. These techniques have implementation and set up costs that are independent of the benchmark and processor complexity. However, statistical sampling may not scale well with longer benchmarks since it requires complete fast-forwarding with warmup simulation (at low detail) through the benchmark execution. Innovations in sample startup techniques (such as checkpointing and direct execution) may ameliorate this overhead for both statistical and representative sampling.

## II.E Summary

Simulating the full reference run of a benchmark in detail is no longer practical. Still, the overall representation of the execution is essential for maintaining a standard in architecture research. A number of techniques have emerged to tackle the increase in benchmark execution and slow simulation time. These techniques are classified in three categories: reduced input, statistical simulation, and sampling. A comparison between these suggests that certain techniques, such as reduced input and statistical simulation, are more appropriate for a large design space search. Sampling techniques are more accurate and should be used for making more refined design decisions. SPEC benchmarks will execute many trillions of instructions in the next suite. It is essential to continue the progression of efficient simulation methodologies to counter the ever increasing benchmark execution length.

# III

# Characterizing Program Behavior and Finding Simulation Points

In this chapter we define the time varying behavior in programs and observe its highly structured nature. We present our method for automatically characterizing this behavior into what we call *phases* and finally describe how using this approach we can find simulation points to accurately represent the entire execution of a program.

## III.A Defining Time Varying Porgram Behavior and Phases

Since phases are a way of describing the recurring behavior of a program executing over time, we begin by describing phase analysis with a demonstration of the time-varying behavior [59] of two programs from the SPEC 2000 benchmark suite, `gcc` and `gzip`. To characterize the behavior of these programs we have simulated their complete execution from start to finish. Each program executes many billions of instructions, and gathering these results took several machine-months of simulation time. The time varying behavior of each program is shown in the Figures III.1 and III.2. The behavior of each program is measured by

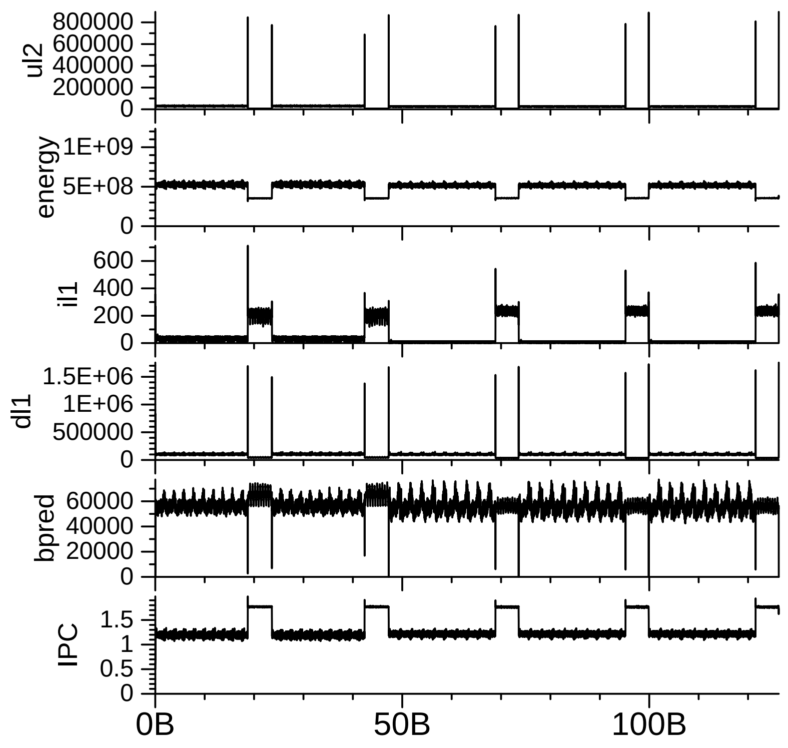


Figure III.1: To illustrate the point that phase changes occur across many metrics all at the same time, we have plotted the value of these metrics over billions of instructions executed for the program `gzip` with input `graphic`. Each point on the graph is an average over 10 million instructions. The number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC are plotted.

various statistics relating to how the program is interacting with the underlying architecture over the course of its execution. The metrics shown are the number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred), and the average IPC.

Each point on the graph represents the average value for that metric (for example cache misses) taken over 10 million instructions worth of execution

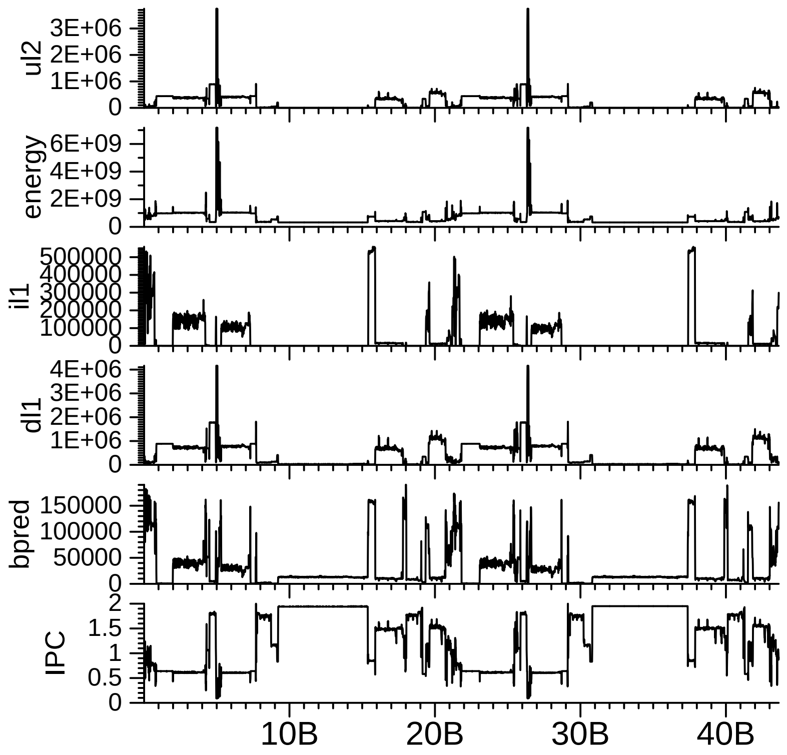


Figure III.2: To illustrate the point that phase changes occur across many metrics all at the same time, we have plotted the value of these metrics over billions of instructions executed for the program `gcc` with input `166`. Each point on the graph is an average over 10 million instructions. The number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC are plotted.

(which we call an interval). There are a couple of important points we can draw from these graphs. The first is that the average behavior does not sufficiently characterize the behavior of the programs. For example, in `gzip` the IPC varies from 1.2 to 1.7 and the number of data cache misses varies by almost an order of magnitude. Note that not only do the behaviors of the programs change over time, they change on the largest of time scales, and even at a large scale one can find repeating behaviors. Programs may have stable behavior for billions of instructions and then change suddenly. The final and most important point to take away from these graphs is that, while the behavior of the program changes significantly over time, the behavior of all of the metrics tend to change in unison, though not necessarily in the same direction. This means that we may be able to extract phase information about how a program changes behavior at a level more general than any particular hardware metric.

The underlying methodology used in this work is the ability to automatically identify these underlying program changes *without relying on architectural metrics*. To ground our discussion in a common vocabulary, the following is a list of definitions to describe program behavior and its automated classification.

- Interval – To perform our analysis we break a program's execution up into non-overlapping intervals of execution. An interval is a section of contiguous execution (a time slice) of a program's execution. For example, when using an interval size of 100 million instructions, the first interval of execution starts at instruction 0 and ends at the 100 million instruction executed, the second interval of execution are the instructions 100 million up to 200 million in the program's execution, the third interval represents instructions 200 to 300 million, etc. For the results in this chapter all intervals are chosen to be the same length, as measured in the number of instructions committed within an interval. This is usually 1, 10, or 100 million instructions, as used by [52]. We will explore using variable length intervals in Chapter V.

- Similarity – A similarity metric measures the similarity in behavior between two intervals of a program's execution, and is specific to the representation of those intervals.
- Phase – A set of intervals within a program's execution that all have similar behavior, *regardless* of temporal adjacency. A phase may be made up of intervals which are disjoint in time; we would call this a phase with a repeating behavior. A "well-formed" phase should have intervals with similar behavior across various architecture metrics (e.g. CPI, cache misses, branch misprediction). In this thesis we consider the terms 'cluster' and 'phase' to be equivalent.
- Phase Classification – Using machine learning to group intervals from a program/input pair into phases (clusters) with similar behavior.

## III.B The Strong Correlation Between Code and Performance

In this section we describe how we identify phase behavior in an architecture independent fashion.

### III.B.1 Using an Architecture-Independent Metric for Phase Classification

To find program phases, we need a notion of how similar are two different parts of a program's execution. In creating this metric it is advantageous to not rely on hardware-based statistics such as cache miss rates or performance (i.e. CPI), since using one of these metrics can mask some of the underlying behavior which does not influence that particular metric. The resulting phases from using hardware-based metrics will end up grouping intervals together that may appear similar on one architecture, but can have very different performance on another architecture. This is not acceptable, since our goal is to find a set of samples that can be used across an architecture design space exploration, where many of these parameters may change. To address this, we need a metric that is *independent* of any particular hardware-based statistic, but still relates to the fundamental changes in behavior like those shown in the graphs of Figures III.1 and III.2.

An effective way to design such a metric is to base it on the behavior of a program in terms of the code that is executed over time. We have shown that there is a very strong correlation [34] between the set of paths executed in a program and the time-varying architectural behavior observed. The intuition behind this is that the executed code determines the behavior of the program. With this idea it is possible to find the phases in programs using *only* a metric related to how the code is being exercised (i.e. both what code is touched and how often). The central idea behind SimPoint is that it can find the phase behavior shown in Figures III.1 and III.2 by examining only the frequency with which the code parts (e.g., basic blocks) are executed over time.

### III.B.2 Basic Block Vector

The basic block vector (BBV) [60] is a structure designed to concisely capture information about how a program is changing behavior over time. A basic block is a section of code (e.g. a contiguous set of instructions) that is executed from start to finish with one entry and one exit. The metric we will use for comparing two time intervals in a program is based on the differences in the execution frequencies for each basic block executed during those two intervals. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing during that interval, and basic block vectors provide us with this information.

A program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides a code signature for that interval of execution, and shows where the application is spending its time in the code. The basic idea is that knowing the basic block distribution for two different intervals gives two separate signatures which we can then compare to find out how similar the intervals are to one another. If the signatures are similar, then the two intervals spend about the same amount of time in the same code, and the performance of those two intervals should be similar.

We represent a basic block vector as a one-dimensional array, with one element in the array for each static basic block in the program. Each interval in an executed program is represented by one BBV, and at the beginning of each interval, its corresponding BBV has all zeros. During each interval, we count the number of times each basic block has been entered, and record that number into the corresponding element in the vector. This number is weighted by the number of instructions in the basic block, since we want every individual instruction to have the same influence. Therefore, each element in the array is the count of how many times its corresponding basic block has been entered during an interval of execution, multiplied by the number of instructions in that basic block. For example, if the 50th basic block has one instruction and is executed 15 times in an interval, then $bbv[50] = 15$ for that interval. At the end of an interval's execution, we normalize the BBV to sum to 1.

We call the vectors used to guide phase analysis *Frequency Vectors*, of which basic block vectors are one type. Frequency vectors can represent basic blocks, branch edges, or any other type of program related structure which provides a representative summary of a program's behavior for each interval of execution. Lau et al. recently examined frequency vector structures other than basic block vectors for the purpose of phase classification. They looked at frequency vectors for data, loops, procedures, register usage, instruction mix, and memory behavior [36]. They found that using register usage vectors, which simply counts for a given interval the number of times each register is defined and used, provides similar accuracy to using basic block vectors. In addition, using only loop and procedure branch execution frequencies performs almost as well as using the full basic block information. For SPEC 2000 programs, Lau et al. [36] found that creating frequency vectors by including both code and data access patterns into the vectors did not improve classification over just using code.

### III.B.3 Basic Block Vector Difference

In order to find patterns in a program we must first have some way of comparing the similarity of two basic block vectors. The operation should take two basic block vectors and return a single number corresponding to how similar (or different) they are.

There are several ways of measuring the similarity of two vectors, such as taking the dot product between the vectors, finding the Euclidean (2-norm) distance of the connecting vector, or Manhattan (1-norm) distance of the connecting vector. The Euclidean distance has been shown to be effective for off-line phase analysis [61, 52]. The SimPoint approach we examine uses Euclidean distance as the metric for comparing basic block vectors, since it is based on $k$-means. For on-the-fly phase analysis (e.g. predicting phases during computation), the Manhattan distance is more efficiently implemented in hardware. It has been shown to be useful in previous work in online phase prediction [62, 37].

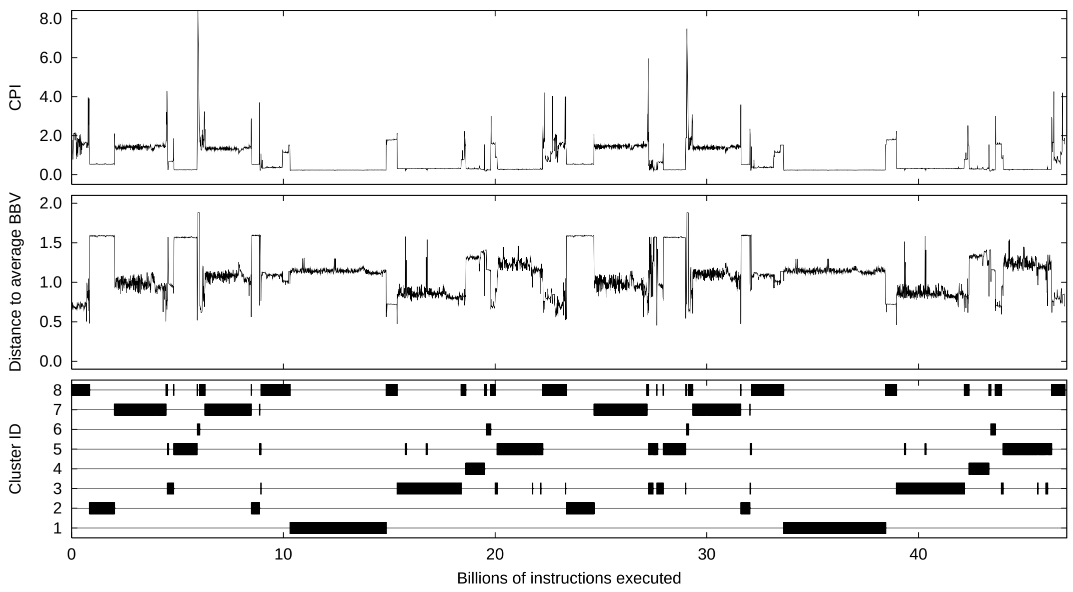


Figure III.3: These plots show the relationship between measured performance (CPI) and code usage for the program `gcc-166`, and SimPoint's ability to capture phase information by only looking at what code is being executed. For each of the three plots, the horizontal axis represents the execution of the program over time, and each point plotted represents one 10-million instruction interval. The top plot shows the CPI for the executing program. The middle plot shows the distance of each interval's basic block vector to the "target vector", which is a basic block vector (explained in Section III.B) that represents the entire program's execution. The target vector is a signature of the program's overall average behavior, and this plot shows how similar the code of each part of the program is to the overall behavior of the program, lower meaning more similar. The bottom plot shows how SimPoint classifies each interval into one of eight phases. The phase transitions correspond to changes in the CPI in the top graph, though SimPoint does not use metrics like CPI to classify intervals.

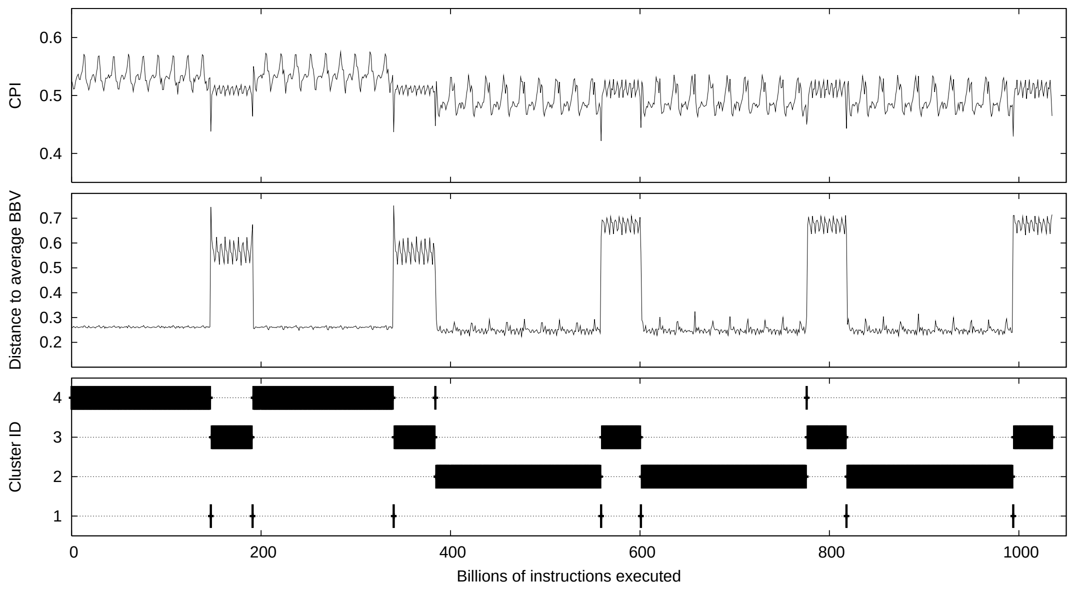


Figure III.4: These plots show the relationship between measured performance (CPI) and code usage for the program `gzip-graphic`, and SimPoint's ability to capture phase information by only looking at what code is being executed. For each of the three plots, the horizontal axis represents the execution of the program over time, and each point plotted represents one 10-million instruction interval. The top plot shows the CPI for the executing program. The middle plot shows the distance of each interval's basic block vector (explained in Section III.B) to the "target vector", which is a basic block vector that represents the entire program's execution. The target vector is a signature of the program's overall average behavior, and this plot shows how similar the code of each part of the program is to the overall behavior of the program, lower meaning more similar. The bottom plot shows how SimPoint classifies each interval into one of four phases. The phase transitions correspond to changes in the CPI in the top graph, though SimPoint does not use metrics like CPI to classify intervals.

### III.B.4 Showing the Correlation Between Code Signatures and Performance

In [34] we demonstrated that there is a strong correlation between executed code and real performance. We showed this using the Receiver Operating Characteristic [51] statistical metric, where we found that changes in CPI and changes in code signature exhibited a strong correlation. The top two graphs of Figure III.3 give one illustration of this correlation by showing the time-varying CPI and BBV distance graphs next to each other for `gcc-166`. The top graph plots the CPI for each interval executed (at 10M interval length) showing how the program's CPI varies over time. Similarly, the BBV distance graph plots for each interval the Manhattan distance of the BBV (code signature) for that interval from the whole program's target vector. The whole program's target vector is a BBV that comes from viewing the whole program as a single interval. The same information is also provided for `gzip` in the top two graphs of Figure III.4. These graphs show that changes in CPI have corresponding changes in code signatures, which is one indication of strong phase behavior for these applications.

These graphs show a strong correlation between code changes and CPI changes even for complex programs like `gcc`. The graphs for `gzip` show that phase behavior can be found even if the intervals' CPIs have small variance. This brings up an important point about classifying intervals based on code similarity rather than based on similarity of CPI or some other hardware metric. Assume we have two intervals with *different code signatures* but they have very *similar CPIs* because both of their working sets fit completely in the cache. During a design space exploration search, as the cache size changes, their CPIs may differ dramatically if one of them no longer fits into the cache. This is why it is important to perform the phase analysis by comparing the code signatures independent of the underlying architecture. We have found that the BBV code signatures correctly identify differences like these, which cannot be seen by looking at just the CPI.

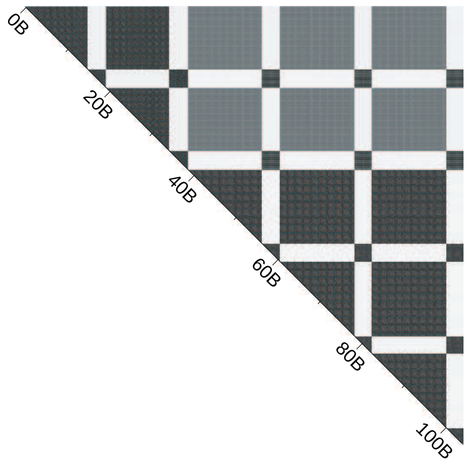
### III.B.5 Basic Block Similarity Matrix

Now that we have methods of comparing program execution intervals, we can use them for finding phase-based behavior. A phase of program behavior can be defined in several ways. Past definitions were built around the idea of a phase being a contiguous interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency. Thus, a phase may appear several times in the execution of a program.

A key observation from this thesis is that the phase behavior seen in any program metric is a function of the code being executed. Because of this we can use the comparison between the basic block vectors to get an idea of how closely related any other metrics will be between those two intervals.

To find how all intervals of execution relate to one another we create a *basic block similarity matrix* for a program/input pair. The similarity matrix is an upper-triangular $n \times n$ matrix, where $n$ is the number of intervals in the program's execution. An entry at $(x, y)$ in the matrix represents the Manhattan distance between the basic block vector at interval $x$ and the basic block vector at interval $y$.

Figures III.5 (left and right) and III.6 (left) shows the similarity matrices for `gzip`, `bzip`, and `gcc` using the Manhattan distance. The diagonal of the matrix represents the program's execution over time from start to completion. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter they are the more different they are (the Manhattan distance is closer to the maximum value — which is 2 since each vector is



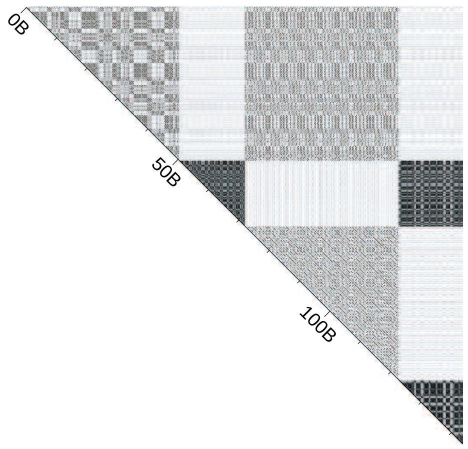
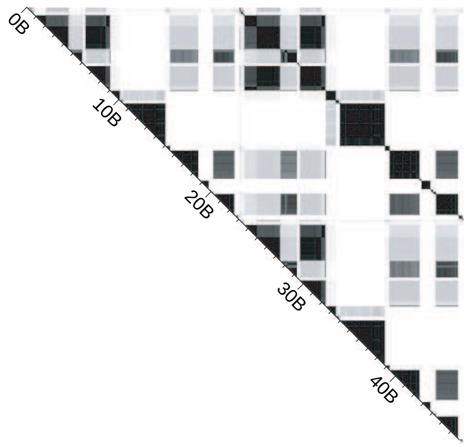


Figure III.5: Basic block similarity matrix for the programs `gzip-graphic` (shown left) and `bzip-graphic` (shown right). The diagonal of the matrix represents the program's execution from beginning to end, with units in billions of instructions. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter the points the more different they are (the Manhattan distance is closer to 2).

normalized to sum to 1).

Consider the points along the matrix diagonal. The top left corner of each matrix is the start of program execution $(0, 0)$, and the bottom right is the point $(n-1, n-1)$ (end of execution). Each interval is perfectly similar to itself, so the points on the diagonal are all dark. Starting from a point on the diagonal, you can compare how its corresponding interval relates to its neighbors forward (backward) in execution by tracing horizontally (vertically) from that point. For example, to compare a given interval $x$ with the interval at $x + m$, start at the point $(x, x)$ on the matrix and trace to the right until you reach $(x, x + m)$.

Let us first examine `gzip` because it has behaviors that are evident at such a large scale that they are easy to see. An interval taken from 70 billion instructions into execution in Figure III.5 (left) is directly in the middle of a large phase shown by the triangle of dark points that surround this point. This means



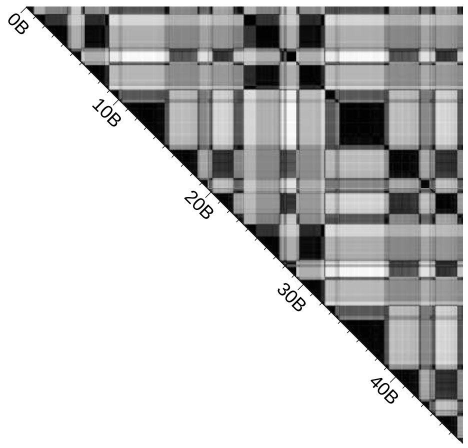


Figure III.6: The original similarity matrix for the program `gcc-166` (left), and the similarity matrix for the projection of `gcc-166` (right). The figure on the left uses the original basic block vectors (each of which has over 100,000 dimensions), and uses the Manhattan distance for calculating the difference. The figure on the right uses the same data, but projected down to 15 dimensions, and uses the Euclidean distance for calculating the difference.

that this interval is very similar to its neighbors both forward and backward in time. We can also see that the intervals at 50 billion and 90 billion instructions are also very similar to the program behavior at 70 billion instructions. While it may be hard to see in a printed version, the intervals around 70 billion instructions are similar to the intervals around 10 billion and 30 billion instructions, and even more similar to those around 50 and 90 billion instructions.

Overall, Figure III.5 (left) shows that the phase behavior seen in the similarity matrix lines up quite closely with the behavior of the program seen in the top graph of Figure III.4, with 5 large regions of self-similar behavior (the first 2 being different from the last 3) each divided by a small region of self-similar behavior. All of the small self-similar regions are also very similar to each other.

The similarity matrix for `bzip` (shown on the right of Figure III.5) is very interesting. `Bzip` has complicated behavior, with two large parts to its execution: compression and decompression. This can readily be seen in the figure as the large dark triangular and square patches. The interesting thing about `bzip` is that even within each of these sections of execution there is complex behavior. This, as will be shown later, makes the behavior of `bzip` impossible to capture using only one small contiguous section of execution.

An even more complex case for finding phase behavior is `gcc`, which is shown on the left of Figure III.6 ( the matrix on the right of that figure will be explained in more detail in Section III.C.1). The left matrix shows that `gcc` does have regular behavior. Even for such a complex program, we see that there is common code shared between sections of execution, such as the intervals around 13 billion instructions and 36 billion instructions. In fact the strong dark diagonal line cutting through the matrix indicates that there is large-scale repetition between the first half and second half of the program. By analyzing the graph we can see that code at each interval $x$ is very similar to interval ($x$+23.6B instructions).

## III.C Automatically Finding Phase Behavior

In this section we describe the algorithms used to automatically detect patterns using the frequency vectors described in the previous section.

### III.C.1 Using Clustering for Phase Classification

A primary goal of SimPoint is to have an automated way of extracting phase information from programs. Data clustering algorithms from unsupervised machine learning have been shown to be very effective at breaking the complete execution of a program into phases that have similar frequency vectors [61]. Because the frequency vectors correlate to the overall performance of the program, grouping intervals based on their frequency vectors produces phases that are similar not only in the distribution of program structures used, but also in every other architecture metric measured, including overall performance.

The goal of clustering is to divide a set of points into clusters such that points within each cluster are similar to one another (by some metric), and points in different clusters are different from one another. We use the machine learning term ‘cluster’ and the architecture term ‘phase’ to express the same concept.

The $k$-means algorithm [41] is an efficient and well-known clustering algorithm, which we use to split program intervals into phases. Prior to clustering, we use random linear projection [13] to reduce the dimension of the input vectors. One drawback of the $k$-means algorithm is that it requires the number of clusters $k$ as an input to the algorithm, but we do not know beforehand what value is appropriate. To address this, we run the algorithm for several values of $k$, and then use a penalized likelihood score to guide our final choice for $k$. Taken to the extreme, if every interval of execution is given its very own cluster, then every cluster will have homogeneous behavior. Our goal is to choose a clustering with a minimum number of clusters that still models the program behavior well.

The following steps summarize the SimPoint phase clustering algorithm at a high level.

1. Profile the program by dividing the program’s execution into contiguous intervals of fixed length (e.g., 1 million, 10 million, or 100 million instructions). For each interval, collect a frequency vector tracking the program’s use of some program structure (basic blocks, branch edges, loops, register usage, etc.). Each frequency vector is normalized so that the sum of all the elements equals 1.

2. Reduce the dimensionality of the frequency vector data to a much smaller number of dimensions using random linear projection. Using projected data speeds up the $k$-means algorithm significantly and reduces the memory requirements by several orders of magnitude while preserving the essential

similarity information.

3. Run the $k$-means clustering algorithm on the projected data with values of $k$ in the range from 1 to $K$, where $K$ is a user-prescribed maximum number of phases that can be detected. Each run of $k$-means produces a clustering, which is a partition of the data into $k$ different phases/clusters. Each run of $k$-means begins with a random initialization step, which requires a random seed.

4. To compare and evaluate the different clusters formed for different $k$, we use the Bayesian Information Criterion (BIC) as a measure of the "goodness of fit" of a clustering to a dataset. A high BIC score indicates the clustering is a good fit to the data. For each clustering ($k \in \{1, 2, \ldots, K\}$), the fitness of the clustering is scored using the BIC.

5. The final step is to choose the clustering with a small $k$ such that its BIC score is nearly as good as the best observed. The chosen clustering is the final grouping of intervals into phases.

The above algorithm groups intervals into phases. This algorithm has several important parameters: interval length, projected dimension, the maximum number of clusters $K$, how the BIC is to be used to select the best clustering, etc. Each must be tuned to create accurate and representative simulation points using SimPoint. We discuss these parameters in more detail later in this thesis.

### Random Projection

For this clustering problem, we have to address the problem of high dimensionality. Many clustering algorithms suffer from the so-called "curse of dimensionality," which refers to the fact that finding an optimal clustering is in-tractable as the number of dimensions increases. For basic block vectors, the number of dimensions is the number of executed basic blocks in the program, which ranges from 2,756 to 102,038 for the SPEC benchmark suite, and could grow into the millions for very large programs. For example, one Microsoft application we studied consisted of over 800,000 basic blocks, which is representative of desktop applications. Another practical problem is that the running time and memory requirements of $k$-means depend on the dimension of the data, making the algorithm slow if the dimension grows too large. Also, we observe that $k$-means tends to get stuck easily in sub-optimal solutions if the dimension is too high. This is evidenced by the small number of iterations $k$-means requires to converge on high-dimensional data, as we have observed on this data. The algorithm does not improve much over its initialization.

Two broad methods of reducing the dimension of data are dimension selection and dimension reduction. Dimension selection simply removes some of the dimensions, based on a measure of goodness of each dimension for describing the data. However, this can throw away a lot of information in the ignored dimensions. Also, in finding a measure to select useful dimensions is not as clear for unsupervised learning as for supervised learning. Dimension reduction reduces the number of dimensions by creating a new lower-dimensional space and then projecting each data point into the new space (where the new space's dimensions are not necessarily related to the old space's dimensions).

For this work we use random linear projection [13] to create a new low-dimensional space into which we orthogonally project the data. This is a simple and fast technique that is very effective at reducing the number of dimensions while retaining the essential structure of the data. There are two steps to projecting a dataset down to a lower-dimensional version. Consider a dataset $X$, which is represented as a matrix of $n \times d$ real values, where $n$ is the number of vectors, and $d$ is the original dimension. We want a low-dimension version $X'$, which is $n \times d'$, where $d'$ is the projected number of dimensions. To create $X'$, we do the following:

- Create a projection matrix $P$ size $d \times d'$. Fill each entry in the matrix with a random value chosen uniformly in $[-1, 1]$.
- Use a matrix multiplication to obtain $X' = X \times P$.

The analysis given by Dasgupta [13] shows that when using random linear projection for clustering data, there are two primary theoretical benefits. The first is that clusters that are very eccentric will become more spherical in their low-dimensional representation. This is appropriate for the $k$-means algorithm which searches for spherical clusters. The second is that a mixture of $k$ Gaussian clusters can be projected into only $O(\log k)$ dimensions while retaining the approximate level of separation between clusters.

Principal components analysis (PCA) is a widely-used method for dimension reduction based on directions of high variance. However, performing PCA on a $d$-dimensional dataset requires $O(d^3)$ operations, which is too expensive for datasets of the size we are considering here that can have hundreds of thousands of dimensions. Constructing the random projection matrix requires only $O(dd')$ time, so it is linear in the original and the new dimension. Dasgupta further showed that there are many simple examples where PCA is not able to reliably reduce $k$ well-separated Gaussian clusters to below $\Omega(k)$ dimensions and keep them well-separated in the low-dimensional projection.

For our application, we found that 15 dimensions is low enough to be computationally tractable, but sufficiently high to discover the different phases of execution with clustering. We found this by running experiments reported in [61]. These experiments projected all the datasets we are interested in to a varying number of dimensions and then recorded the number of clusters found by

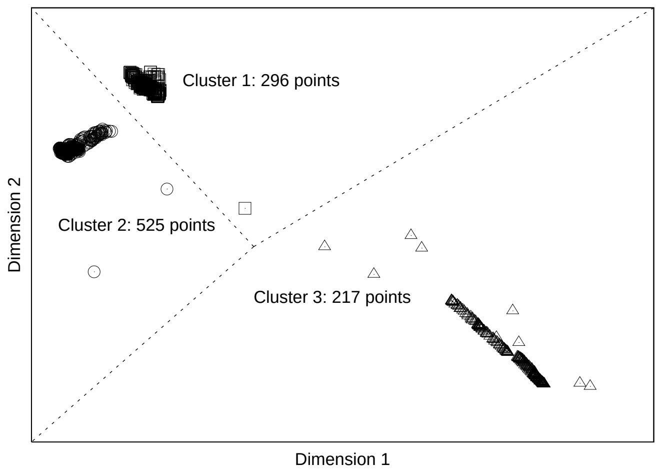


Figure III.7: This plot shows a two-dimensional projection of the basic block vectors for the program `gzip`, having 1038 total intervals, and clustered into three clusters with $k$-means. The lines show divisions between the three clusters. Note that SimPoint normally operates in more than two dimensions, but this illustrates the fact that that program behavior does form natural groups that can be found through data clustering.

$k$-means and the BIC. We found that for fewer than 15 dimensions, the number of clusters found dropped off, but for more than 15 dimensions, the number of clusters found did not increase significantly.

Figure III.6 shows the similarity matrix for `gcc` on the left using original BBVs, whereas the similarity matrix on the right shows the same matrix but on the data that has been projected down to 15 dimensions. For the reduced dimension data we use the Euclidean distance to measure differences. Some information is lost because of the projection, but overall phase behavior we see in the original data is still easily discernible with only 15 dimensions. A scatterplot of the program `gzip` projected to 2 dimensions and clustered into 3 clusters using $k$-means is shown in Figure III.7.

**Bayesian Information Criterion**

To compare the different clusterings formed for different $k$, we use the Bayesian Information Criterion, or BIC [58], as a measure of the "goodness of fit" of a clustering to a dataset. The BIC is an approximation of the probability of the clustering, given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering being scored is a "good fit" to the data being clustered. The BIC formulation we use is appropriate for clustering with $k$-means, however other formulations of the BIC could also be used for other clustering models. The BIC is only one method of choosing a good model from a set of models; other methods such as the Akaike information criterion (AIC) [1], minimum description length (MDL) [57], and Monte-carlo cross-validation (MCCV) [65] may also be appropriate.

There are two parts of the BIC: the likelihood and the penalty. The likelihood is a measure of how well the clustering models the data. For the $k$-means likelihood, each cluster's model is considered a spherical Gaussian distribution (which is the assumption $k$-means makes). The likelihood of a cluster is the product of the probabilities of each point in the cluster given by the cluster's Gaussian. The likelihood for the whole model is just the product of the likelihoods for all clusters. However, the likelihood tends to increase without bound as more clusters are added. Therefore the second term is a penalty that offsets the likelihood growth based on the model complexity (i.e. the number of clusters). The BIC is formulated as

$$BIC(X, C_k) = \mathcal{L}(X|C_k) - \frac{p}{2}\log(n)$$

where $\mathcal{L}(X|C_k)$ is the log-likelihood of the clustered data $X$ given the clustering $C_k$ having $k$ clusters, $n = |X|$ is the number of points in the data, and $p = (k-1)+dk+1 = k(d+1)$ is the number of parameters to estimate: $(k-1)$ cluster probabilities, $k$ cluster center estimates which each requires $d$ mean estimates, and one variance estimate (shared over all clusters). The log-likelihood of the $k$-means model given the data is

$$\mathcal{L}(X|C_k) = -\frac{nd}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{k}\sum_{i\in C_j}||X_i - c_j||^2 + \sum_{j=1}^{k} n_j \log(n_j/n)$$

where $n_j$ is the number of points in the $j$th cluster (so $n_j/n$ is the estimated prior probability of cluster $j$), and $\sigma^2$ is the average squared Euclidean distance from each point to its cluster center. The term $C_j$ represents the set of all indexes of $X$ that are members of cluster $j$, $X_i$ is the $i$th point in dataset $X$, and $c_j = \frac{1}{n_j}\sum_{i\in C_j} X_i$ is the location of the $j$th cluster center. The center $c_j$ is the maximum likelihood solution for the cluster's center. The maximum likelihood estimator for $\sigma^2$ is:

$$\hat{\sigma}^2 = \frac{1}{nd}\sum_{j=1}^{k}\sum_{i\in C_j}||X_i - c_j||^2$$

For the purposes of calculating the BIC, we can substitute this maximum likelihood estimate for $\sigma^2$ into the log-likelihood formulation, to get a simpler version:

$$\mathcal{L}(X|C_k) = -\frac{nd}{2}\log(2\pi\sigma^2) - \frac{nd}{2} + \sum_{j=1}^{k} n_j \log(n_j/n)$$

The BIC formulation we present basically follows that given by [50].

For a given program and inputs, the BIC score is calculated for each $k$-means clustering, for $K$ in the range 1 to $K$. We then choose the clustering that achieves a BIC score that is close to the highest BIC score seen.

### III.C.2 Clusters and Phase Behavior

The bottom plots in Figures III.4 and III.3 show the results of running our phase-finding clustering algorithm on `gzip` and `gcc`. These results use an interval length of 10 million instructions and the maximum number of phases ($K$) is set to 10. The horizontal axis corresponds to the execution of the program (in billions of instructions), and each interval is classified to belong to one of the clusters (labeled on the vertical axis).

For `gzip`, the program's execution is partitioned into 4 clusters. Looking at the middle plot for comparison, the cluster behavior captured by our algorithm lines up quite closely with the behavior of the program. Clusters 2 and 4 represent the large sections of execution which are similar to one another. Cluster 3 captures the smaller phase that lies in-between these larger phases. Cluster 1 represents the phase transitions between the three dominant phases. The intervals in cluster 1 are grouped into the same phase because they execute a similar combination of code, which happens to be part of the code behavior in either cluster 2 or 4 and part of code executed in cluster 3. These transition points in cluster 1 also correspond to the same intervals that have large spikes in CPI seen in the top graph (these spikes are due to increased cache misses for those regions).

The bottom plot of Figure III.3 shows how `gcc` is partitioned into 8 clusters. Comparing this to the middle and top plots in the same figure, we see that even the more complicated behavior of `gcc` is captured well by SimPoint. The dominant behaviors in the top two graphs can be seen grouped together in phases 1, 3, 5,and 7.

## III.D Methodology

For this study, we performed our analysis for the complete set of SPEC CPU2000 programs for multiple inputs using the Alpha binaries from the SimpleScalar website. We collect all of the frequency vector profiles, described in Section III.B, using SimpleScalar. To generate our baseline results, we executed all programs from start to completion using SimpleScalar, gathering the hardware metrics. The baseline microarchitecture model is detailed in Table III.1.

Table III.1: Baseline Simulation Model.

| | |
|---|---|
| I Cache | 16k 2-way set-associative, 32 byte blocks, 1 cycle latency |
| D Cache | 16k 4-way set-associative, 32 byte blocks, 2 cycle latency |
| L2 Cache | 1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency |
| Main Memory | 150 cycle latency |
| Branch Pred | hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor |
| O-O-O Issue | out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer |
| Mem Disambig | load/store queue, loads may execute when all prior store addresses are known |
| Registers | 32 integer, 32 floating point |
| Func Units | 8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV |
| Virtual Mem | 8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete |

To examine the accuracy of our approach we provide results in terms of CPI prediction error. The CPI prediction error is the percent difference between CPI predicted using only simulation points chosen by SimPoint and the baseline (true) CPI of the complete execution of the program. The CPI error is computed in the following manner:

$$\text{CPI Error} = \frac{|\text{True CPI} - \text{SimPoint Estimated CPI}|}{\text{True CPI}}$$

## III.E Choosing Simulation Points from the Phase Classification

After the phase classification algorithm has done its job, intervals with similar code usage will be grouped together into the same phases (clusters). Then from each phase, SimPoint chooses one representative interval that will be simulated in detail to represent the behavior of the whole phase. Therefore, by simulating *only* one representative interval per phase, we can extrapolate and capture the behavior of the entire program.

To choose a representative for a cluster, SimPoint picks the interval that is closest (Euclidean distance) to the cluster's $k$-means center. The center can be viewed as a pseudo-interval which behaves most like the average behavior of the entire phase. Most likely there is no interval that exactly matches the center, so SimPoint chooses the closest interval. The selected interval is called a *simulation point* for that phase [52, 61]. We can then perform detailed simulation on the set of simulation points.

As part of its output SimPoint also gives a weight for each simulation point. Each weight is a fraction: it is the total number of instructions represented by the intervals in the cluster from which the simulation point was taken divided by the number of instructions in the program. With the weights and the detailed simulation results of each simulation point, we can compute a weighted average for the architecture metric of interest (CPI, cache miss rate, etc.) for the entire program's execution.

These simulation points are chosen once for a program/input combination because they are chosen based only on how the code is executed, and not based on architecture metrics. Therefore, they only need to be calculated once for a binary/input combination and can be used repeatedly across all of the runs for an architecture design space exploration.

The number of simulation points that SimPoint chooses has a direct effect on the simulation time that will be required for those points. The maximum number of clusters, $K$, along with the interval length, represents the maximum amount of simulation time that will be needed. When fixed length intervals are used, ($K *$ interval length) is a limit on the number of simulated instructions.

SimPoint allows users to trade off simulation time with accuracy. Researchers in architecture tend to want to keep simulation time to below a fixed number of instructions (e.g., 300 million) for a run. If this is a goal, we find that an interval length of 10 million instructions with $K = 30$ provides very good accuracy (as we show later in this thesis) with reasonable simulation time (220 million instructions on average). If even more accuracy is desired, then decreasing the interval length to 1 million and setting $K = 300$ performs well for the SPEC 2000 programs, as does setting $K = \sqrt{n}$ (where $n$ is the number of clustered intervals). Empirically we discovered that as the granularity becomes finer, the number of phases discovered increases at a sub-linear rate. The upper bound defined by this square-root heuristic works well for the SPEC benchmarks.

The length of the interval chosen by users of SimPoint depends upon their simulation infrastructure and how much they want to deal with warmup. Warmup is the process of initializing the simulator's state (caches, branch predictor, etc.) at the start of a simulation point so that it is the same as if we simulated from the beginning of the program to that point. For many programs, using a long interval length (e.g., more than 100 million instructions) will make warmup unnecessary. This is the approach used by Intel's PinPoint for simulation [48]. They simulate intervals of length 300-500 million instructions so they do not have to worry about implementing warmup in their simulation infrastructure. With such long intervals the architecture structures are warmed up sufficiently during the beginning of the interval's execution to provide accurate simulation results. In comparison, short interval lengths can be used, but this requires having an approach for warming up the architecture state. One way to do this is with an architecture checkpoint, which stores the potential contents of the major architecture components at the start of the simulation point [5]. This can significantly reduce warmup time, since warmup consists of just reading the checkpoint from a file and using it to initialize the architecture structures. It may also be desirable to use only a single sample to represent a program. This would allow for a simpler simulation infrastructure, since only one segment of the execution would need to be simulated in detail.

### III.E.1 Accuracy of SimPoint

Figure III.8 shows the simulation accuracy results using SimPoint (and other methods) for the SPEC 2000 programs when compared to the complete execution of the programs. For these results we use an interval length of 100 million instructions and limit the number of simulation points to no more than 10. With the above parameters SimPoint finds 4 phases for `gzip`, and 8 for `gcc`. As described above, one simulation point is chosen for each cluster, so this means that a total of 400 million instructions were simulated for `gzip`. The results show that this results in only a 4% error in performance estimation for `gzip`.

For these results, we compare this estimated IPC using SimPoint to the baseline IPC. IPC (Instructions Per Cycle) is the inverse of CPI, and often used instead of CPI when describing performance. The baseline was gathered from spending months of simulation time to simulate the entire execution of each SPEC program. The results in Figure III.8 compare SimPoint to how architecture researchers use to choose where to simulate before SimPoint. The first technique was to just simulate the first N million instructions of a benchmark's execution. The second technique was to blindly skip the first billion instructions of execution to get past the initialization of the program's execution, and then simulate for N million instructions. The results show that simulating from the start of execution, for the exact same number of instructions as simulated with SimPoint, results in a median error of 58%. If instead, we fast forwarded for 1 billion instructions and then simulate for the same number of instructions as chosen by SimPoint, we see a median 23% IPC error. When using SimPoint to create multiple simulation points we have a median IPC error of 2%. Note that the maximum error seen for the prior techniques are significant for the SPEC programs, but it is very

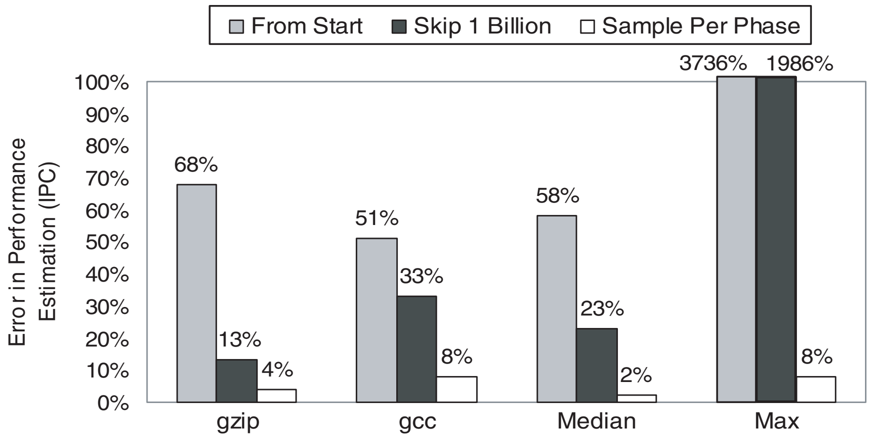


Figure III.8: Simulation accuracy for the SPEC 2000 benchmark suite when performing detailed simulation for several hundred million instructions compared to simulating the entire execution of the program. Results are shown for simulating from the start of the program's execution, for fast-forwarding 1 billion instructions before simulating, and for using SimPoint to choose at most ten 100-million-instruction intervals to simulate. The results are shown as percent error of predicted IPC, which is how much the estimated IPC using SimPoint is different from the complete execution of the program. IPC is the inverse of CPI. The median and maximum results are for the complete SPEC 2000 benchmarks.

reasonable (only 8%) for SimPoint.

### III.E.2 Relative Error During Design Space Exploration

The absolute error of a program/input run on one hardware configuration is not as important as tracking the change in metrics across different architecture configurations. There is a lot of discussion and research into getting lower simulation error rates. But what often is not discussed is that a low error rate for a single configuration is not as important as achieving the same relative error rates across the design space search and having them all biased in the same direction.

We now examine how SimPoint tracks the relative change in hardware metrics across several different architecture configurations. To examine the independence of the simulation points from the underlying architecture, we used the simulation points for the SimPoint algorithm with an interval length of 1 million instructions and the maximum $K$ set to 300. For the program/input runs examined, we performed full program simulations while varying the memory hierarchy, and for every run we used the same set of simulation points when calculating the SimPoint estimates [52].

We varied the configurations and the latencies of the L1 and L2 caches. As we increased the size and the associativity of the caches we increased their latency to model the effect of architecture scaling. We chose L1 instruction and data cache sizes between 4 KBytes and 64 KBytes, varied the associativity between direct mapped and 4-way associative, and varied their latency from 1 to 3 cycles. At the same time we varied the size of the unified L2 cache from 500 KBytes to 2 MBytes, its associativity from 4-way to 8-way associative, and varied the latency from 10 to 40 cycles.

Figure III.9 shows the results across 19 different architecture configura-

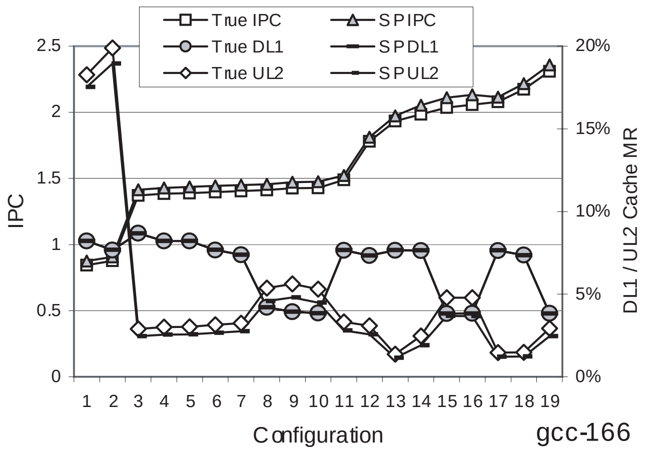


Figure III.9: This plot shows the true and estimated IPC and cache miss rates for 19 different architecture configurations for the program `gcc`. The left $y$-axis is for the IPC and the right $y$-axis is for the cache miss rates for the L1 data cache and unified L2 cache. Results are shown for the complete execution of the configuration and when using SimPoint.

tions for `gcc-166`. The left $y$-axis represents the performance in Instructions Per Cycle (IPC) and the $x$-axis represents different memory configurations from the baseline architecture. The right $y$-axis shows the miss rates for the data cache and unified L2 cache, and the L2 miss rate is a local miss rate. For each metric, two lines are shown: "True" for the true metric from the *complete* detailed simulation, and the "SP" for the estimated metric using our simulation points. For the results, the configurations on the $x$-axis are sorted by the IPC of the full run.

This figure shows that the simulation points, which are chosen by only looking at code usage, can be used across different architecture configurations to make accurate architecture design trade-off decisions and comparisons. The simulation points are able to track the relative changes in performance metrics between configurations. This means we are able to make the same decision between two architectures, in terms of which one is better, using SimPoint as the complete simulation of the program. One interesting observation is that although the simulation results from SimPoint have a bias in its predictions, this bias is consistent across the different configurations for a given program/input. This is true for both IPC and cache miss rates. We believe one reason for the bias is that SimPoint chooses the most representative interval from each phase, and intervals that represent phase change boundaries are less likely to be fully represented across the chosen simulation points.

## III.F Summary

Understanding the cycle level behavior of a processor running an application is crucial to modern computer architecture research, and gaining this understanding can be done efficiently by judiciously applying detailed cycle level simulation to only a few simulation points. By targeting only one or a few carefully chosen samples for each of the small number of behaviors found in real programs, the cost of simulation can be reduced to a reasonable level while achieving very accurate performance estimates.

The main idea behind SimPoint is the realization that programs typically only exhibit a few unique behaviors that are interleaved with one another through time. By finding these behaviors and then determining the relative importance of each one, we can maintain both a high level picture of the program's execution and at the same time quantify the cycle level interaction between the application and the architecture. The key to being able to find these phases in an efficient and robust manner is the development of a metric that can detect the underlying shifts in a program's execution that result in the changes in observed behavior. In this Chapter we have discussed one such method of quantifying executed code similarity, and use it to find program phases through the application of unsupervised learning techniques.

SimPoint automates the process of picking simulation points using an off-line phase classification algorithm based on $k$-means clustering, which significantly reduces the amount of simulation time required. Selecting and simulating only a handful of *intelligently* picked sections of the full program provides an accurate picture of the complete execution of a program, which gives a highly accurate estimate of performance. The SimPoint software can be downloaded at:

`http://www.cse.ucsd.edu/users/calder/simpoint/`

For the industry-standard SPEC programs, SimPoint has less than a 6% error rate (2% on average) for the results in this chapter, and is 1,500 times faster on average than performing simulation for the complete program's execution. Because of this time savings and accuracy, our approach is currently used by architecture researchers and industry companies (e.g. [48] at Intel) to guide their architecture design exploration.

## Acknowledgements

Portions of the text in this chapter are in part a reprint of the material from the paper, *Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood, "Using Machine Learning to Guide Architecture Simulation"*, in the Journal of Machine Learning Research, 2006. The dissertation author was the secondary researcher and author and the co-authors involved in the publication [23] directed, supervised, and assisted in the research which forms the basis for that material.

# IV

# Data Mining and Statistical Advances in Phase Analysis

In this chapter we describe data-mining and statistical advances in doing phase analysis. These advances optimize both the runtime and accuracy performance of SimPoint as well as target the overall simulation time.

First we explore the primary parameters that have influence on how SimPoint and the k-means clustering algorithm behave. We focus on how we achieve a reasonable running time for k-means, and how the dominant parameters affect this runtime. These include the number of intervals to cluster, the dimension of the intervals being clustered, and the number of iterations it takes to perform a clustering. We also present a method to sub-sample the vectors during clustering, which enables the clustering of a very large data set. Additionally, we examine how to search over k to find a good clustering efficiently. We describe a binary search method over k clusterings that achieves comparable accuracy to an exhaustive search, but at a fraction of the time.

Secondly, we present an algorithm for choosing the number of clusters to use by augmenting the BIC with confidence information called *Variance Sim-*

*Point*. We use statistical analysis to guide the picking of a clustering to provide a given level of confidence and probabilistic error bound. In addition, this same analysis can be used as a post step if desired to provide a confidence and error bound for a clustering derived using the original SimPoint BIC algorithm.

The final contribution presented in this chapter deals with picking earlier simulation points to decrease simulation time. For a given clustering, our prior algorithms focus on picking the most representative point from a cluster, since we assumed a simulation environment with check-pointing. Check-pointing is the process of storing the state of a simulator so that simulation can continue from that point at a later time. If the start of all of the simulation points are check-pointed, then the full program can be simulated in parallel very quickly simulating each checkpoint independently. In practice, not all simulation environments have the support for check-pointing, and instead the simulator must fast-forward (perform functional emulation) between the simulation points. This can take a significant amount of time, especially if the simulation point is at the end of execution. Therefore, we created algorithms for picking simulation points that are earlier in the program to significantly reduce the fast-forward time needed for these simulation environments.

## IV.A Clustering Analysis

In this section we describe the primary parameters that have influence on how SimPoint and the $k$-means algorithm behave. We first focus on how we achieve a reasonable running time for $k$-means, and then examine how to search over $k$ to find a good clustering. For the experiments in this section, we use basic block vectors with 100 million instruction intervals. Where it is not specified, we also use $k = 30$ clusters and 15 projected dimensions.

### IV.A.1 Methods for Reducing the Run-Time of $k$-Means

Even though SimPoint only needs to be run once per binary/input combination, we still want a fast clustering algorithm that produces accurate simulation points. To address the run-time of SimPoint, we first look at the three parts that have the largest effect on the running time of a single run of $k$-means. The three parts are the number of intervals to cluster, the dimension of the intervals being clustered, and the number of iterations it takes to perform a clustering.

We first examine how the number of intervals affects the running time of the SimPoint algorithm. Figure IV.1 shows the time (in seconds) for running SimPoint on different numbers of intervals as we vary the number of clusters. For this experiment, the clustered vectors are randomly generated from uniformly random noise in 15 dimensions. We use random data in these experiments because it does not bias these results based on a particular benchmark and it gives comparable results across a wide range of parameter settings. But more importantly, prior theoretical work by [28] suggests that it is most difficult to accelerate clustering algorithms on data without structure, such as uniformly random data. This is supported by experiments by [43] and [17]. So these experiments form a comparable set of challenging results for the per-iteration run-time of SimPoint. The number of iterations will vary depending on the structure of the data, however. For example, using $k$-means to cluster data from very well-separated clusters is likely to converge in a low number of iterations, while clusters that overlap are likely to require more iterations.

The first graph shows that for 100,000 vectors and $k = 128$, it took about 3.5 minutes for SimPoint 3.0 to perform the clustering. It is clear that the number of vectors clustered and the value of $k$ both have a large effect on the run-time of SimPoint. The run-time changes linearly with the number of clusters and the number of vectors, as expected. Also, we can see that the time per basic

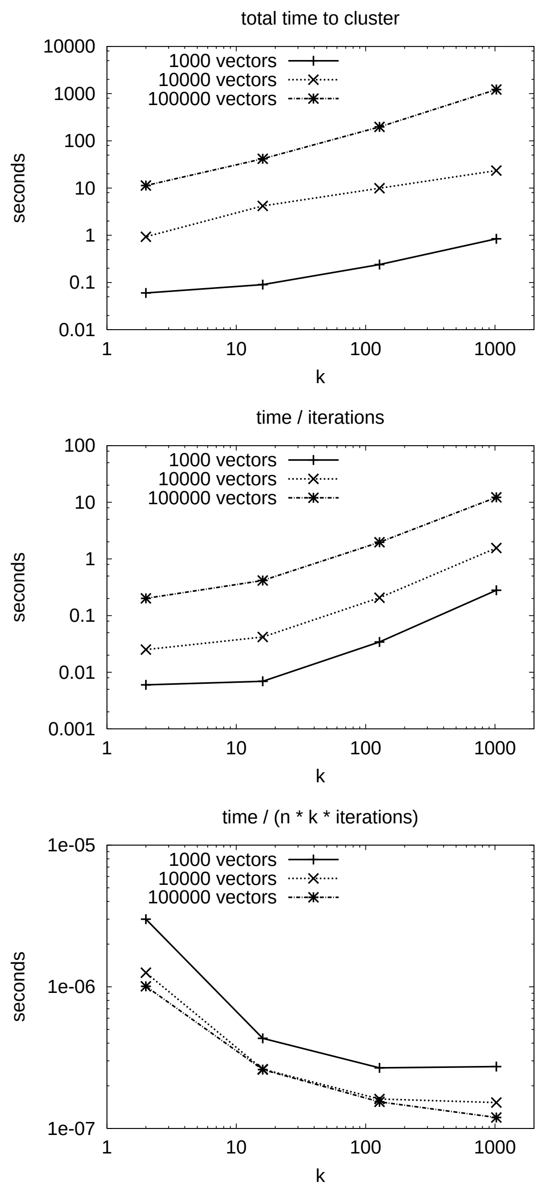


Figure IV.1: These plots show how varying the number of vectors and clusters affects the amount of time required to cluster with SimPoint 3.0. For this experiment we generated uniformly random data in 15 dimensions. The first plot shows total time, the second plot shows the time normalized by the number of iterations performed, and the third plot shows the time normalized by the number of basic operations performed. Both the number of vectors and the number of clusters have a linear influence on the run-time of $k$-means. The bottom plot shows a decreasing trend due to optimizations in $k$-means which are more beneficial for larger $k$.

operation actually goes down as $k$ increases. This is due to a simple optimization called *partial distance search* [42, 10] that allows the algorithm to avoid calculating the full distance from a point (interval) to every cluster center in the first step of $k$-means. The goal of this step is to find the closest cluster center to the point, so that the interval may be assigned to that center. To find this closest center, a simple loop searches for the cluster center with the minimum squared Euclidean distance. The squared distance calculation sums the squared dimension difference between the point and the cluster center over all dimensions. While searching for the minimum squared distance from a point to all centers, partial distance search keeps the smallest squared distance seen thus far. When calculating the distance to another center, it may find that the intermediate squared distance result (after processing some of the dimensions) is larger than the smallest squared distance seen to a different center. If this is the case, the distance we are calculating cannot be minimal, so the current calculation is stopped short of calculating the entire squared distance over all of the dimensions. This optimization does not change the correctness of the algorithm. Partial distance search is most beneficial when there are many clusters, since the more centers there are, the more it is likely that there will be a close center that can give a good lower bound for the partial search. Partial distance search is also useful in high dimensional data, since work is saved when computing per-dimension differences, and the more dimensions there are the more computations can potentially be avoided.

### Number of Intervals and Sub-sampling

Each iteration of the $k$-means algorithm has a run-time that is linear in the number of clusters, the number of intervals, and the dimensionality. However, since $k$-means is an iterative algorithm, many iterations may be required to reach convergence. We found that we can reduce the number of dimensions down

Table IV.1: This table shows the running times (in minutes) by SimPoint 3.0 without using binary search (SP3-All) and SimPoint 3.0 using binary search (SP3-BinS). SimPoint is run searching for the best clustering from $k$=1 to 100, uses 5 random seeds per $k$, and projects the vectors to 15 dimensions. The second column shows how many vectors and the size of the vector (static basic blocks) the programs have.

| Program | # Vecs × # B.B. | SP3-All | SP3-BinS |
|---|---|---|---|
| gcc-166 | 4692 × 102038 | 9 min | 3.5 min |
| crafty | 19189 × 16970 | 84 min | 10.7 min |

to 15 and still maintain SimPoint's accuracy. Therefore, the main influence on execution time for SimPoint is the number of intervals.

To show this effect, Table IV.A.1 shows the SimPoint running time for `gcc-166` and `crafty-ref`, which shows the lower and upper limits for the number of intervals and basic block vectors seen in SPEC 2000 with an interval length of 10 million instructions. The second and third columns show the number of intervals and original number of dimensions for each basic block vector. The last two columns show the time it took to execute SimPoint 3.0 searching for the best clustering from $k$=1 to 100, with 5 random initializations (seeds) per $k$. The fourth column shows the time it took to run SimPoint when searching over all $k$, and the last column shows clustering time when using a binary search described in Section IV.A.2. The results show that increasing the number of intervals by 4 times increased the running time of SimPoint around 10 times. The results also show that the number of intervals clustered has a large impact on the running time of SimPoint, since it can take many iterations to converge, which is the case for `crafty`. We used 15 dimensions during clustering for these results.

The effect of the number of intervals on the running time of SimPoint becomes critical when using very small interval lengths like 1 million instructions or fewer, which can create millions of intervals to cluster. To speed the execution of SimPoint on these very large inputs, we sub-sample the set of intervals that will be clustered, and run $k$-means on only this sample. To sample with SimPoint, the user specifies the number of desired interval samples, and then SimPoint chooses that many intervals (without replacement). The probability of each interval being chosen is proportional to the weight of its interval (the number of dynamically executed instructions it represents). For vectors that represent the same interval length (as we consider in this chapter), this weight is uniform. If vectors represent non-uniform interval lengths (called variable-length intervals, or VLIs), then each vector's weight is proportional to its interval length. We describe our work with variable length intervals in Section V.A.

Sampling is common in clustering for datasets that are too large to fit in main memory [18, 56]. After clustering the dataset sample, we have a set of clusters with centers found by $k$-means. SimPoint then makes a single pass through the unclustered intervals and assigns each interval to the cluster that has the nearest center (centroid) to that interval. This then represents the final clustering from which the simulation points are chosen.

The experiments shown in Figure IV.2 show the effects of sub-sampling across all the SPEC 2000 benchmarks using an interval length of 10 million instructions, 30 clusters, and 15 projected dimensions. Results are shown for creating the initial clustering using sub-sampling with only 1/8, 1/4, 1/2, and all of the execution intervals in each program, as described above. The first two plots show the effects of sub-sampling on the CPI errors and $k$-means variance, both of which degrade gracefully when smaller samples are used. The average SPEC INT (integer) and SPEC FP (floating point) results are shown. It is standard to break the results into these two groupings for architecture results.

The average $k$-means variance is the average squared distance between every frequency vector and its closest cluster center. Lower variances are better.

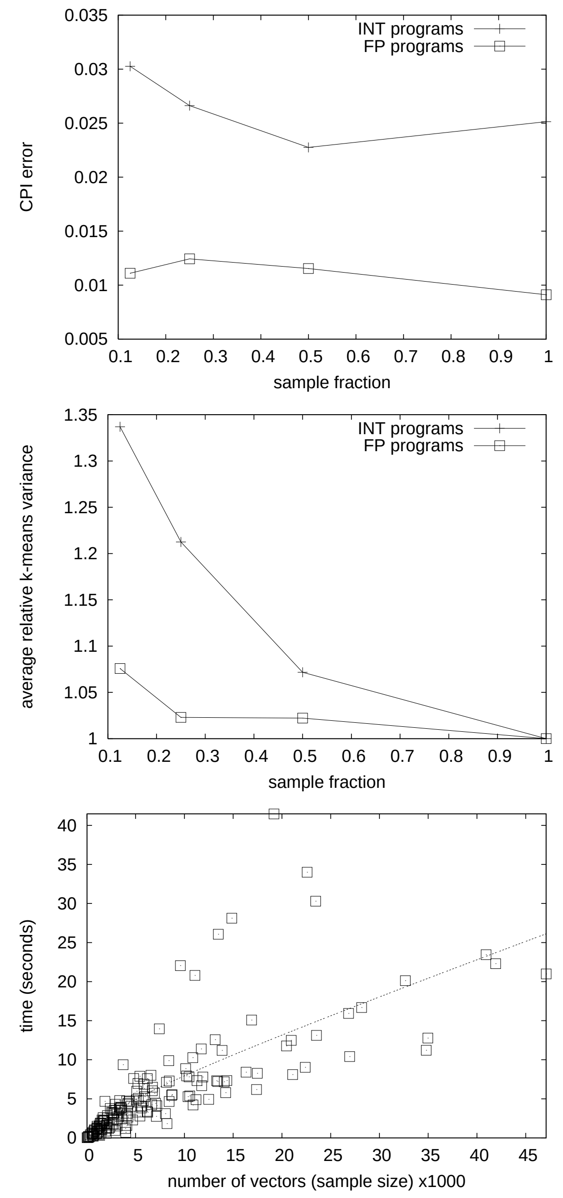
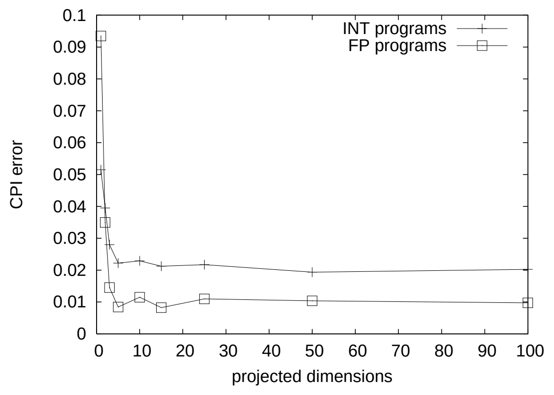


Figure IV.2: These three plots show how sub-sampling before clustering affects the CPI errors, $k$-means variance, and the run-time of SimPoint. The first plot shows the average CPI error across the integer and floating-point SPEC benchmarks. The second plot shows the average $k$-means clustering variance relative to clustering with all the vectors. The last plot shows a scatter plot of the run-time to cluster the full benchmarks and sub-sampled versions, and a logarithmic curve fit with least squares.

When sub-sampling, we still report the variance based on every vector (not just the sub-sampled ones). The *relative* $k$-means variance reported in the experiments is measured on a per-input basis as the ratio of the $k$-means variance for clustering on a sample to that of clustering on the whole input.

As shown in the second graph of Figure IV.2, sub-sampling a program can cause $k$-means to find a slightly less representative clustering, which results in higher $k$-means variance on average. Note that the $k$-means variance for these experiments are reported on all the input vectors, not just the sampled ones. Even so, when sub-sampling, we found in some cases that it can reduce the $k$-means variance and/or CPI error (compared to using all the vectors), because sub-sampling can remove outliers in the dataset that $k$-means may be trying to fit. This is a benefit noted in the work of [19] when they use sub-sampling to initialize iterative clustering algorithms.

It is interesting to note the difference between floating point and integer programs, as shown in the first two plots. The results shown in the first plot show we can capture the behavior of the SPEC floating point programs more easily, that is, without using all the original data. In addition, the second plot suggests that SPEC floating point programs are also easier to cluster than the SPEC INT, as we can do quite well (in terms of $k$-means variance) even with only small samples. This suggests that they have more regular or uniform code usage patterns than integer programs. The third plot shows the effect of the number of vectors on the running time of SimPoint. This plot shows the time required to cluster all of the benchmark/input combinations and their 3 sub-sampled versions. In addition, we have fit a logarithmic curve with least-squares to the points to give a rough idea of the growth of the run-time. Note that two different datasets with the same number of vectors may require different amounts of time to cluster due to the number of $k$-means iterations required for the clustering to converge.



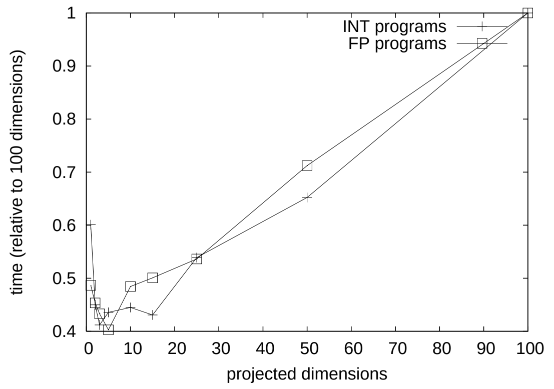


Figure IV.3: These two plots show the effects of changing the number of projected dimensions when using SimPoint. The default number of projected dimensions SimPoint uses is 15, but here we show results for 1 to 100 dimensions. The left plot shows the average CPI error, and the right plot shows the average time relative to 100 dimensions. Both plots are averaged over all the SPEC 2000 benchmarks, for a fixed $k = 30$ clusters.

### Number of Dimensions and Random Projection

Along with the number of vectors, the other most influential aspect in the running time of $k$-means is the number of dimensions of the data. Figure IV.3 shows the effect of changing the number of projected dimensions on both the CPI error (left) and the run-time of SimPoint (right). For this experiment, we varied the number of projected dimensions from 1 to 100. As the number of dimensions increases, the time to cluster the vectors increases linearly, as expected. It is more interesting that the run-time also increases for very low dimensions. This is because the points are more “crowded” and the clusters are less well-separated, so $k$-means requires more iterations to converge.

If we use too few dimensions, the data does not retain sufficient information to cluster the data well. This is reflected by the fact that the CPI errors increase rapidly for very low dimensions. However, we can see that at 15 dimensions, the SimPoint default, the CPI errors are quite low, and using a higher number of dimensions does not improve them significantly but requires more computation. Using too many dimensions is also a problem in light of the well-known “curse of dimensionality” [3], which implies that as the number of dimensions increases, the number of vectors that would be required to densely populate that space grows exponentially. This means that using a higher dimension makes it more likely that a clustering algorithm will converge to a poor solution, since the input space is not very densely filled. Therefore, it is wise to choose a dimension that is low enough to allow $k$-means to find a good clustering, but not so low that critical information is lost. We find that 15 dimensions works well in these regards.

### Number of Iterations Needed

The final aspect we examine for affecting the running time of the $k$-means algorithm is the number of iterations it takes for a run to converge. We provide this analysis to illustrate typical requirements of running SimPoint on a set of benchmarks, and because finding a tight upper-bound on the number of iterations required by $k$-means is an open problem [14], we must rely on evidence to show us what to expect.

The $k$-means algorithm iterates either until it hits a user-specified maximum number of iterations, or until it reaches convergence. In SimPoint, the

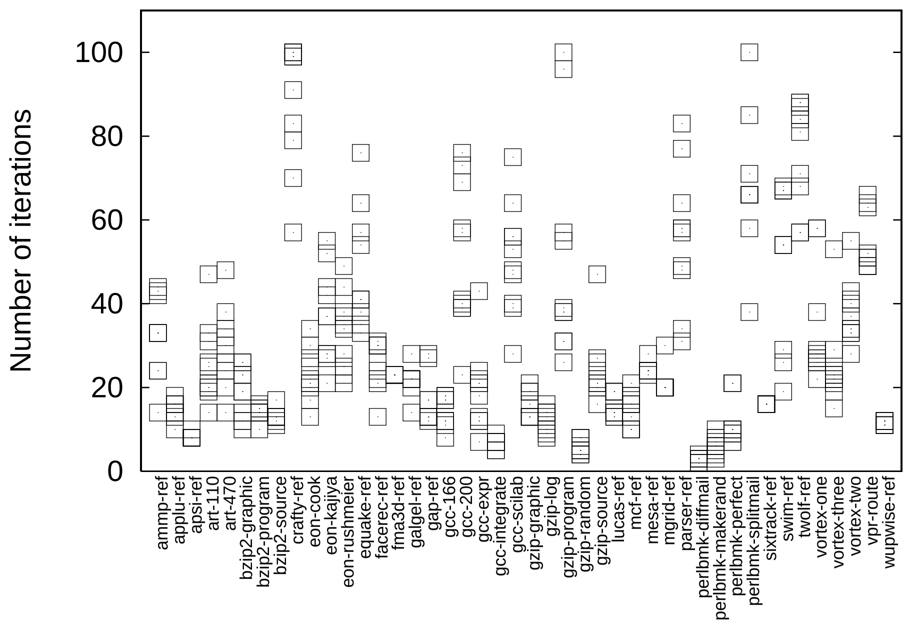


Figure IV.4: This plot shows the number of iterations required for 10 randomized initializations of each benchmark, with 10 million instruction length intervals, $k = 30$, and 15 dimensions. Note that only three program/inputs had a total of 5 runs that required more than the default limit of 100 iterations, and these all converge within 160 iterations or less.

default limit is 100 iterations, but this can easily be changed. More iterations may be required, especially if the number of intervals is very large compared to the number of clusters. The interaction between the number of intervals and the number of iterations required is the reason for the large SimPoint running time for `crafty-ref` in Table IV.A.1.

For our results, we observed that only 1.1% of all runs on all SPEC 2000 benchmarks reach 100 iterations. This experiment was with 10-million instruction intervals, $k$=30, 15 dimensions, and with 10 random initializations of $k$-means. Figure IV.4 shows the number of iterations required for all runs in this experiment. Out of all of the SPEC program and input combinations run, only `crafty-ref`, `gzip-program`, `perlbmk-splitmail` had runs that had not converged by 100 iterations. The longest-running clusterings for these programs reached convergence in 160, 126, and 101 iterations, respectively. If desired, SimPoint can always run $k$-means to convergence (with no iteration limit).

### IV.A.2 Searching for a Small $k$ with a Good Clustering

We suggest setting the maximum number of clusters $K$ as appropriate for the maximum amount of simulation time a user will tolerate for a single simulation. SimPoint uses three techniques to search over the possible clusterings, which we describe here. The goal is to try to pick a small $k$ so that the number of simulation points is also small, thereby reducing the simulation time required.

#### Setting the BIC Percentage

As we examine several clusterings and values of $k$, we need to have a method for choosing the best clustering. The Bayesian Information Criterion (BIC) [50] gives a score of how well a clustering represents the data it clustered. However, we have observed that the BIC score often increases as the number of

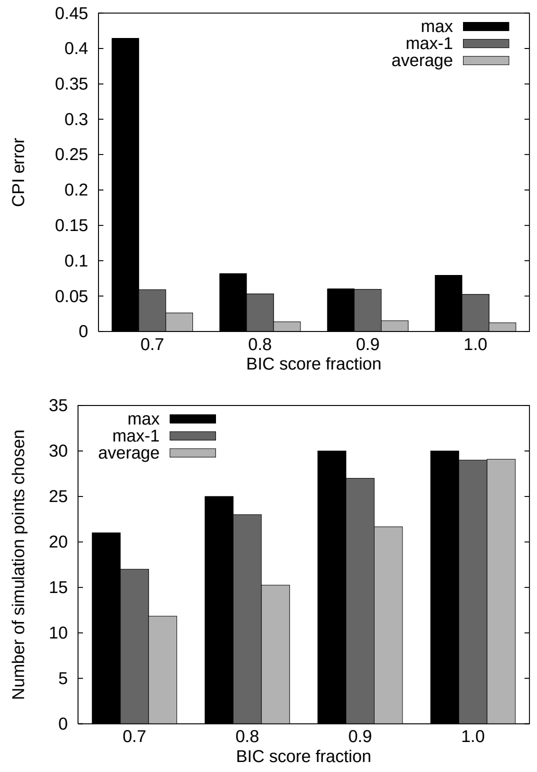


Figure IV.5: These plots show how the CPI error and number of simulation points chosen are affected by varying the BIC threshold. Bars labeled "max-1" show the second largest value observed.

clusters increase. Thus choosing the clustering with the highest BIC score can lead to often selecting the clustering with the most clusters. Therefore, we look at the range of BIC scores, and select the score that attains some high percentage of this range. The SimPoint default BIC threshold is 90%. When the BIC rises and then levels off as $k$ increases, this method chooses a clustering with the fewest clusters that is near the maximum BIC value. Choosing a lower BIC threshold would prefer fewer clusters, but at the risk of less accurate simulation.

Figure IV.5 shows the effect of changing the BIC threshold on both the CPI error (left) and the number of simulation points chosen (right). These experiments are for using binary search (explained in Section IV.A.2) with $K =$

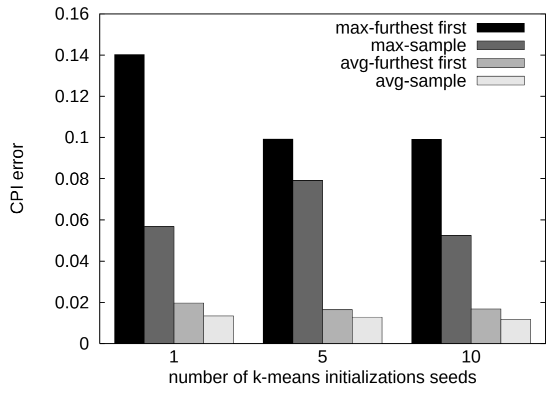


Figure IV.6: This plot shows the average and maximum CPI errors for both sampling and furthest-first $k$-means initializations, and using 1, 5, or 10 different random seeds. These results are over the SPEC 2000 benchmark suite for 10-million instruction vectors, 15 dimensions, and $k = 30$.

30, 15 dimensions, and 5 random seeds. BIC thresholds of 70%, 80%, 90% and 100% are examined. As the BIC threshold decreases, the average number of simulation points decreases, and similarly the average CPI error increases. At the 70% BIC threshold, `perlbmk-splitmail` has the maximum CPI error in the SPEC suite. This anomaly is an artifact of the low threshold. Since higher BIC scores point to better clusterings and better error rates, we recommend the BIC threshold to be set at 90%.

### Varying the Number of Random Seeds, and $k$-means initialization

The $k$-means clustering algorithm starts from a randomized initialization, which requires a random seed. Because of this, running $k$-means multiple times can produce very different results depending on the initializations, so $k$-means can sometimes converge to a locally-good solution that is poor compared to the best clustering on the same data for that number of clusters. Therefore, conventional wisdom suggests that it is good to run $k$-means several times using a different randomized starting point each time, and take the best clustering observed, based on the $k$-means variance or the BIC. SimPoint does this, using different random seeds to initialize $k$-means each time. Based on our experience, we have found that using 5 random seeds works well.

SimPoint allows users to provide their own $k$-means initialization, or it will choose an initialization based on one of two methods: sampling and furthest-first [21, 26]. The sampling method chooses $k$ random locations for the initial cluster centers from the input data without replacement. The furthest-first method chooses one input point at random, and then repeatedly chooses a point that is furthest away from all the already-chosen points, until $k$ points are chosen. This has the tendency to spread the initially chosen points out along the convex hull of the input space, and subsequently chosen points in the interior.

Figure IV.6 shows the effect on CPI error of using two different $k$-means initialization methods (furthest-first and sampling) along with different numbers of initial $k$-means seeds. These experiments are for using binary search with $K = 30$, 15 dimensions, and a BIC threshold of 90%. When multiple seeds are used, SimPoint runs $k$-means multiple times with different starting conditions and takes the best result.

Based on these results we see that sampling outperforms furthest-first $k$-means initialization. This can be attributed to the data we are clustering, which can have a large number of outlying points, which furthest-first initialization pays special attention to. The furthest-first method is likely to pick those anomaly points as initial centers since they are the furthest points apart. It is also beneficial to try multiple seed initializations in order to avoid a locally minimal solution. The results in Figure IV.6 shows that 5 seed initializations should be sufficient in finding good clusterings.

### Binary Search for Picking $k$

SimPoint 3.0 makes it much faster to find the best clustering and simulation points for a program trace over earlier versions. Since the BIC score generally increases as $k$ increases, SimPoint 3.0 uses this knowledge to perform a binary search for the best $k$. For example, if the maximum $k$ desired is 100, with earlier versions of SimPoint one might search in increments of 5: $k = 5, 10, 15, \ldots, 90, 100$, requiring 20 clusterings. With the binary search method, we can ignore large parts of the set of possible $k$ values and examine only about 7 clusterings.

The binary search method first clusters 3 times: at $k = 1$, $k = K$, and $k = (K + 1)/2$. It then proceeds to divide the search space and cluster again based on the BIC scores observed for each clustering and the user-specified BIC threshold. Thus the binary search method requires the user only to specify the maximum number of clusters $K$, and performs at most $\log_2(K)$ clusterings.

Figure IV.7 shows the comparison between the new binary search method for choosing the best clustering, and the old method, which searched over all $k$ values in the same range. The top graph shows the CPI error for each program, and the bottom graph shows the number of simulation points (clusters) chosen. These experiments are for using binary search with $K = 30$, 15 dimensions, 5 random seeds, and a BIC threshold of 90%. Exhaustive search performs slightly better than binary search, since it searches all $k$ values. Using the binary search, it possible that it will not find a clustering with as few clusters as found by the exhaustive search. This is shown in the bottom graph of Figure IV.7, where the exhaustive search picked 19 simulation points on average, and binary search chose 22 simulation points on average. In terms of CPI error rates, the average is about the same across the SPEC programs between exhaustive and binary search. Recall that the binary search method operates many times faster than the brute

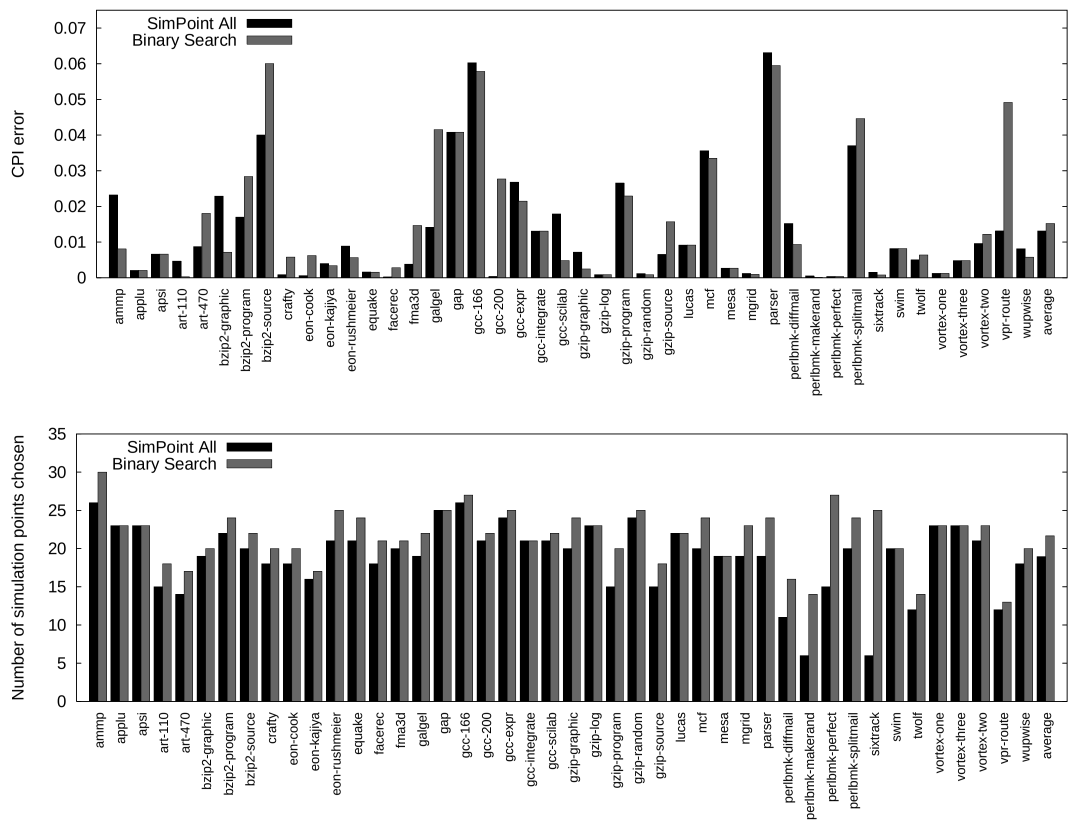


Figure IV.7: These plots show the CPI error and number of simulation points chosen for two different ways of searching for the best clustering. The first method, which was used in SimPoint 2.0, searches over all $k$ between 1 and 30, and chooses the smallest clustering that achieves the BIC threshold of 90%. The second method is the binary search for $K = 30$, which examines at most 5 clusterings.

force search method (see Table IV.A.1 for some timing results).

As we can see from the graphs in Figure IV.7, SimPoint is able to achieve a 1.5% CPI error rate averaged across all SPEC 2000 benchmarks, with a maximum error of around 6%. These results require an average simulation time of about 220 million instructions per program (for the binary search method). These error rates are sufficiently low to make design decisions, and the simulation time is small enough to do large-scale design space explorations.

## IV.B Picking Simulation Points Using Statistical Analysis

In this section we first show how to find a probabilistic error bound for a given level of confidence for a single set of simulation points. This can be used to estimate the confidence and error for a given clustering. We then show how to use this same analysis to create a new algorithm for picking $k$ (the number of clusters to use). We then compare the results of this algorithm with statistical sampling.

### IV.B.1 Statistical Validation of Simulation Points

Given a set of simulation points derived from SimPoint, a user may want to know the confidence and error for this set of simulation points. We use the following approach for estimating what the expected error is for a given level of statistical confidence for a single set of simulation points. We then show how to use this technique to guide picking $k$ (the cluster to use) for SimPoint.

To establish the error bounds, we must find the variance of the estimator. A very simple and intuitive way to do this is to repeatedly take estimates from the program. This sampling technique is known as a parametric bootstrap [8, p. 480], where the parametric form is the clustering structure we have learned.

Recall from Section III.E that for the clustering $k$ chosen by SimPoint, one simulation point (interval) is chosen to represent the entire cluster. This set of simulation points for all the clusters is then used to represent the complete execution of the program. To find the error bound for one set of simulation points, we do the following:

1. Find a clustering using the BIC heuristic
2. Do the following $N$ times:
    (a) Choose one interval (sample) at random from each cluster.
    (b) Compute an estimated CPI by combining the CPIs from each chosen interval, weighted by the size of each cluster.
3. The probabilistic error bound on *one* set of simulation points is $z\sigma/\mu$.

Here $\mu$ is the average of the N estimated CPIs, and $\sigma$ is the standard deviation of the computed estimate CPIs. The value $z$ is the "confidence multiplier" that comes from a table of the normal distribution; $\alpha = F(z)$, where $F$ is the cumulative distribution function of the normal distribution, so that $z = F^{-1}(\alpha)$. Thus, $z$ is the value such that the area under the Gaussian curve to the left of $z$ is $\alpha$, the desired confidence. Then $e$ is the probabilistic error bound for a given level of confidence $\alpha$, where $e$ is $z\sigma/\mu$. For a choice of *one* set of simulation points, we expect the true CPI to be within $z\sigma/\mu$ of the estimate $\mu$.

The value of $N$ is the number of times to compute estimates. Larger $N$ gives a more accurate and tighter measurements of the standard deviation; $N = 10$ or larger is reasonable; for our measurements we have used $N = 100$. Note that gathering all the simulation points for the $N$ CPI estimates only requires one run through the program. We first choose $N$ random samples from every cluster, and then run the program once fast-forwarding between the samples to gather all of the results. Then, the above analysis can be used to determine for a given confidence level a probabilistic error bound for any SimPoint clustering.

### IV.B.2 Picking K using Variance Analysis

We now describe a new SimPoint algorithm where the user enters a desired confidence and a probabilistic error bound, and then the smallest clustering $k$ is picked that matches these constraints. The first priority of the algorithm is to ensure that candidate clusterings are chosen *first* according to the homogeneity of their clusters based on code usage, and then *second* based upon a confidence and probabilistic error bound. This is because the confidence and error are calculated with respect to CPI and sampling a particular architecture configuration. If we did not choose a clustering based first upon code usage and instead only on confidence and error, then the clustering may not be representative across different hardware configurations.

In this algorithm we cluster the data for all possible values of $K$ from 1 to max $K$ that is specified by the user. To ensure that a clustering is picked that would be representative and independent of the underlying architecture we first apply the BIC heuristic to all of the clusterings. The new algorithm starts to differ here. We trim down the possible set of clusterings from $K$ down to $B$. These $B$ clusterings have a BIC score greater than a specified threshold (80% for the results in this section). We then search this candidate set of $B$ clusterings for the smallest $k$ that meets the desired confidence and error. Picking a subset of clusterings based on BIC and then a final clustering based on Variance in this manner ensures that the given set of simulation points chosen will be representative of the complete execution regardless of the underlying architecture.

Our Variance SimPoint algorithm allows the user to calibrate the accuracy that is expected for a given clustering. For example, the error bounds can be set to be within 5% of the true value with a 95% confidence level. The algorithm uses sampling to determine an appropriate number of clusters for that desired error.

1. Choose a desired error $e$ and confidence level $\alpha$.

2. For each $k = \{1, \ldots, K\}$, find a clustering of the Basic Block Vectors with $k$ clusters using the $k$-means algorithm. Here $K$ is the maximum number of clusters to consider, also specified by the user.

3. Trim $K$ down to $B$ candidate clusterings. Candidate clusterings are those with a BIC score that is at least 80% as good as the best BIC score. This insures that we only examine clusterings that are well formed, by first only considering the similarities in the code executed between intervals. This results in a set $B$ of *candidate clusters* and only these clusters will be considered in the rest of the algorithm.

4. From *each candidate clustering*, choose $N$ samples randomly *from each cluster*. This results in a total of $S$ samples to gather.

5. Run the program/input pair gathering architecture results for each of the $S$ samples.

6. For each candidate clustering $k$, calculate the estimated CPI $\mu_k$ the standard deviation $\sigma_k$, and error as described above in Section IV.B.1. For the desired confidence level $\alpha$, the probabilistic error $e$ is calculated to be $e = z\sigma_k/\mu_k$.

7. Select the smallest $k$, from the set of candidate clusterings $B$ such that $z\sigma_k/\mu_k \leq e$. This picks the smallest $k$ that has a small enough standard deviation to satisfy the desired error bound at the given confidence level.

The above algorithm determines a set of candidate clusterings $B$ that satisfy the above BIC heuristic. This ensures that we are only considering clusterings that are well formed. To perform the Variance analysis, we need to gather samples from the candidate clusterings. For each clustering in $B$, we randomly select $N$ points from each cluster to sample. For example, assume we start with a max $K = 20$, then after applying the BIC score we are left with only three candidate clusterings, which are $k = 18, 19, 20$ (remember that $k = 18$ has 18 clusters, $k = 19$ has 19 clusters, and so on). We then have to gather $(18 + 19 + 20) * N$ samples to calculate the confidence and error analysis for each of these three clusterings. Once the sample locations are chosen, the program/input pair is then simulated once to gather all of the samples (fast-forwarding between samples and detailed simulation for each sample). Once this is done, we can then calculate the probabilistic error bound for the given input confidence level for each clustering $k$. We then choose the smallest candidate $k$ such that $z\sigma_k/\mu_k \leq e$, using the statistical analysis described above in Section IV.B.1.

For some programs it may be that for the max $K$ used (limit on the number of clusters) that $\sigma_k$ never reaches a small enough value to allow any clusterings to be acceptable at the desired error and confidence. In this case, several options can be followed. One option is to increase the maximum $K$ to consider, and repeat the above process, which will cause the $\sigma_k$ to decrease for the new, larger possible values of $k$. Another option is to use the clustering $k$ with the lowest $\sigma_k$, understanding that the desired error will not be achieved; rather, the expected error will be within $c\sigma_k/\mu_k$ percent of the true value.

### IV.B.3 Variance SimPoint Results

Figure IV.8 shows the results for the Variance SimPoint algorithm using an interval (sample) size of 1 million instructions, and a max $K$ of 300 clusters, so at most 300 million instructions would be detailed simulated. The desired confidence level used for these results is for $\alpha = 0.95$, or 95% confidence with an error less than 5%. For this method, we gathered results for 43 SPEC program/input combinations. Figure IV.8 shows the results for all of these for the Variance SimPoint algorithm. The percent error is shown when compared to the complete execution of the program/input, and top of the stem shows the probabilistic error bound.

Figures IV.9 and IV.10 show the number of clusters used and the number of instructions the program had to fast-forward through to get to the last simulation point chosen for the Variance SimPoint results in Figure IV.8. In these two figures, detailed results are shown for a subset of 10 programs, `avg-rest` shows the average for all the 33 programs not shown in detail, and then an `avg-all` shows the overall average for all 43 programs. The results show that on average 100 million instructions are simulated in detail, and there is usually a simulation point chosen somewhere near the end of the program, so the fast-forward distance is almost equal to the full execution of the program.

The second and third bars in Figure IV.8 show results for random sampling with (1) the same number of intervals chosen as for the Variance SimPoint algorithm, and (2) when using 1000 samples. The sample size used is also 1 million instructions. For these results, we use a form of random sampling instead of systematic sampling. To perform our random sampling, we divide the program up into N consecutive sections, where N is the number of samples we are going to take. We then randomly choose one point from each of the N sections. This guarantees that we get a random distribution of samples across the complete execution of the program/input. For the program/inputs examined, we found this to provide tighter probabilistic error bounds and lower errors than pure random sampling or systematic sampling.

The results show that Variance SimPoint is able to achieve tighter probabilistic error bounds than Sampling, when using the same number of samples. Variance SimPoint ensure that at least one sample is being used from each cluster, and the purpose of clustering is to group program behavior such that each cluster potentially represents different program behavior. Therefore, obtaining a

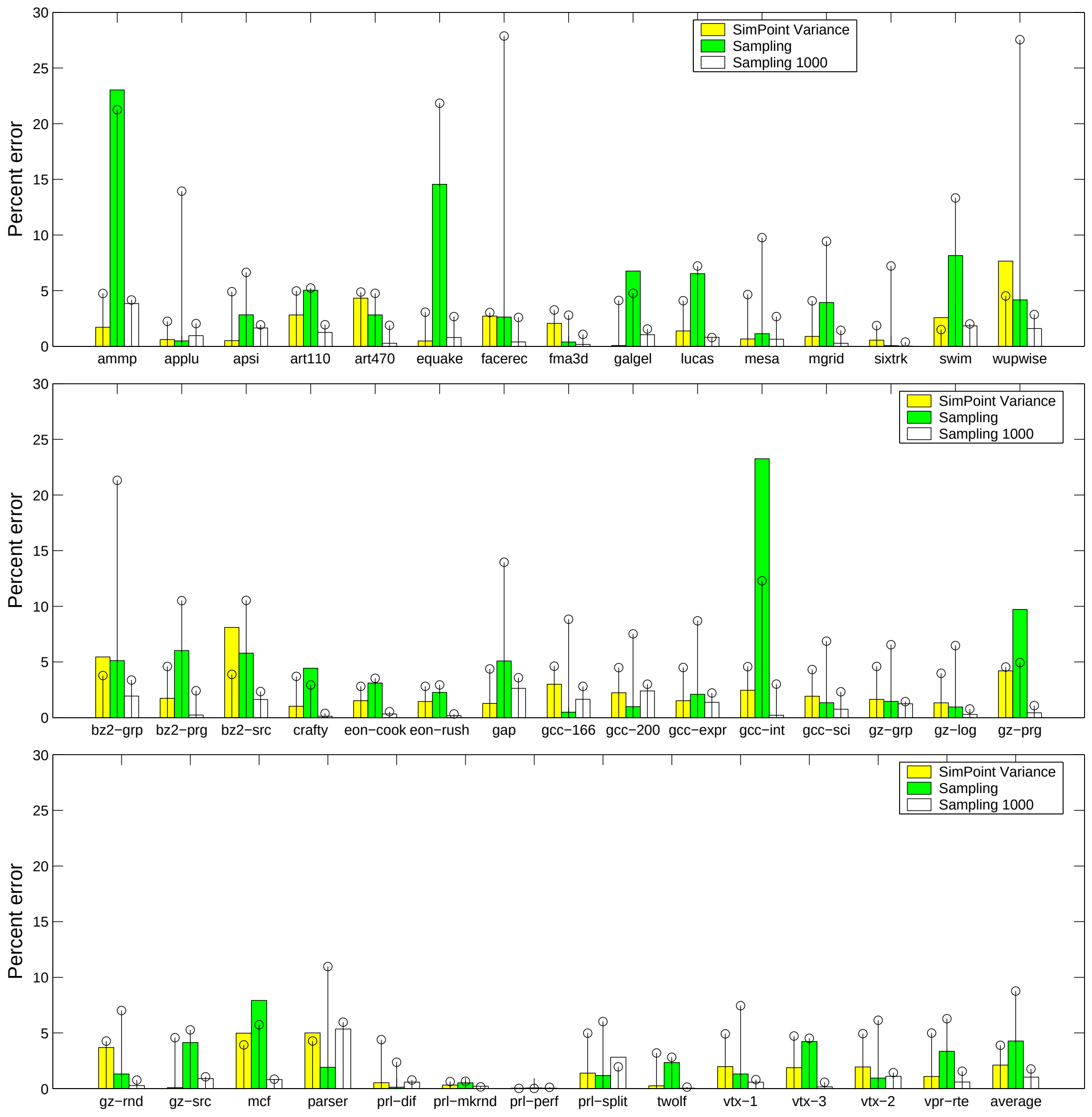


Figure IV.8: True CPI error (bars) and probabilistic error bounds (stems) for SimPoint using variance to choose $k$ (the clustering). Variance SimPoint is compared to random sampling using the same number of samples as SimPoint, and random sampling with 1000 samples. The probabilistic error bounds are for 95% confidence.

sample of all of the different behaviors in a program allows Variance SimPoint to achieve tight error bounds when using a small number of samples (on average 100 as shown in Figure IV.9). In comparison, when using the same number of samples for each program, random sampling has a worse probabilistic error bound. This is because random sampling can oversample some of the program's behavior while under-sampling other parts of the program. When using 1000 samples, less than a 2% average error is seen with under a 3% probabilistic error bound.

## IV.C Early Simulation Points

In the standard SimPoint algorithm [61], the goal is to pick a single simulation point from each cluster that best represents all of the intervals in that cluster. This may pick simulation points that are at the very end of a program's execution. If the simulator supports checkpointing, then simulation can be started very quickly at a point at the end of the program. But, for simulation environments that do not support checkpointing, it can require up to several days to fast-forward to the latter part of execution to reach a late simulation point.

The goal in this section is to find simulation points that are earlier in the program's execution that still accurately represent the overall execution of the program. These early simulation points can then be used to significantly reduce the time spent fast-forwarding to reach all of the simulation points for a program.

### IV.C.1 Early SimPoint Algorithm

This section focuses on a simulation environment that relies upon using fast-forwarding to simulate a program. In this environment, the program is simulated once interleaving fast-forwarding with detailed simulation. The *last* simulation point in a program's execution determines how much of the program the simulator will have to fast-forward through, and this greatly determines the total simulation time. Therefore, to reduce the time required for fast-forwarding, we only care about the location of the last simulation point. If we pick the earliest point from each cluster, then the earliest the last simulation point in the program will occur is the location of the latest starting cluster.

Our Early SimPoint algorithm focuses on choosing a clustering that is both representative of the program's execution and has some feasible simulation points early in the program for all clusters. This might not be achievable for all programs, since an important phase of execution may only appear at the end of execution. We therefore still give priority in our algorithm to ensure that the clustering groups together intervals of execution that are similar to one another. Once a clustering is chosen, we pick representative simulation points early in the execution from all the clusters.

## Picking a Clustering

As described in Chapter III, the standard SimPoint algorithm uses $k$-means to perform several clusterings for different values of $k$. It then uses the BIC score to choose a clustering, and then from each cluster the point nearest the centroid is picked to represent that cluster. For the Early SimPoint algorithm, we perform the exact same $k$-means clustering algorithm as in the original SimPoint algorithm. The Early algorithm differs in how it chooses which clustering to use, and then how it selects a representative point from each cluster.

In picking a cluster ($k$), the Early SimPoint algorithm takes into consideration where the intervals for a given cluster are located over time (the execution of the program). The goal is to pick a clustering, where all clusters have some intervals early in the program's execution, while still clustering together similar

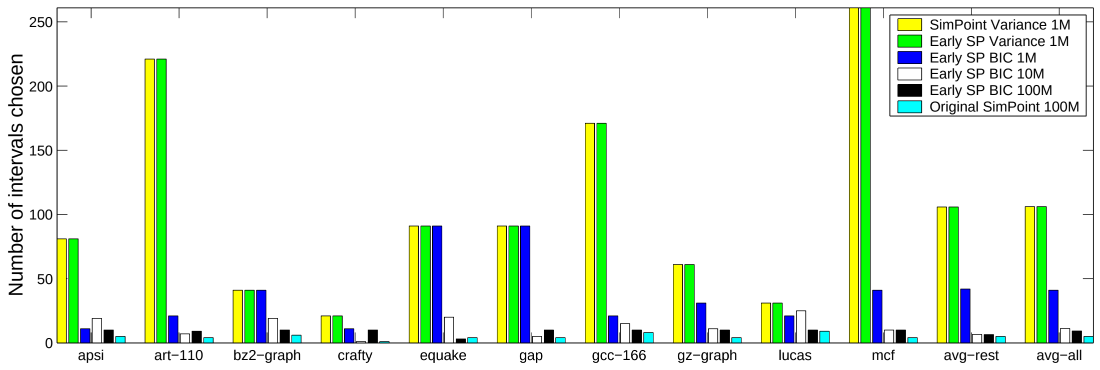


Figure IV.9: Number of simulation points used for each program/input for the different SimPoint algorithms. The number of simulation points is equivalent to the value of $k$ chosen, which is the number of clusters. A single simulation point is picked from each cluster.

intervals of execution.

To guide this we introduce a new metric, *EarlySP*, which is the BIC score weighted by the first encounter of the last cluster: $EarlySP = BIC \times (1 - (StartLastCluster/w))$. The intuition behind *EarlySP* is that we reward the clusterings that have representatives from every cluster near the start of the program. $StartLastCluster$ is the percent into execution of the program that the last cluster is first encountered. For example, if for a clustering we have a $StartLastCluster$ of 40%, this means that one of the clusters has its first interval of execution occurring 40% into execution, and all of the rest of the clusters have at least one interval earlier in the execution than this. The $StartLastCluster$ is critical for picking early simulation points, since it will be the minimum distance required to fast-forward.

The variable $w$ is a weight to influence the impact of how early the last cluster is on the BIC. Since we prioritize accurate program representation over early simulation, the variable $w$ limits how much the $BIC$ can be influenced by the $StartLastCluster$ term. Setting $w$ to 10 guarantees that we remain within 10% of the true $BIC$, ensuring representative scoring is maintained for the clusterings.

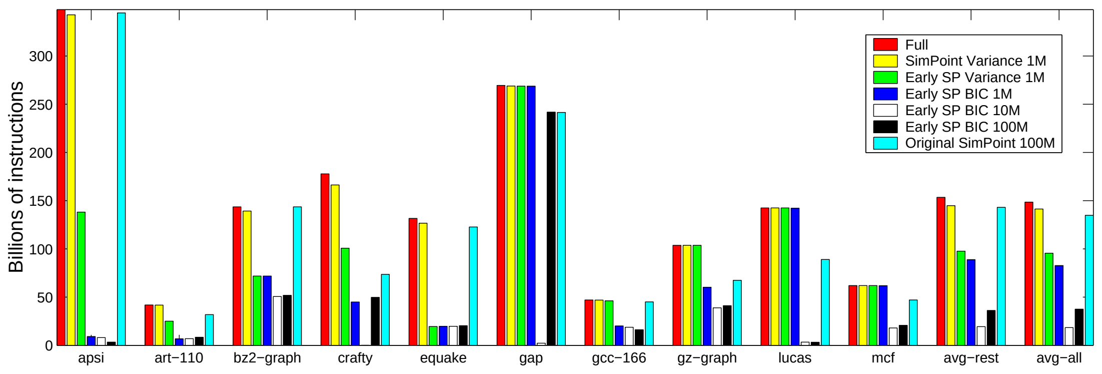


Figure IV.10: The number of instructions in Billions needed for fast-forwarding to reach the last simulation point for the different SimPoint algorithms. The results for Full show the length of the complete execution of each program/input run. The goal of Early SimPoint is to reduce the amount of fast-forwarding by picking representative, but early, simulation points.

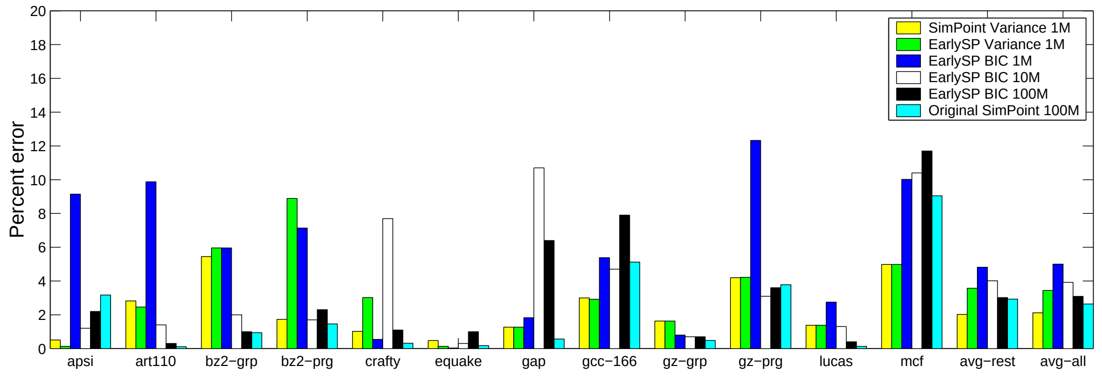


Figure IV.11: Relative CPI error to the full simulation for the different SimPoint algorithms varying the interval size over 1, 10, and 100 Million sample sizes.

The *EarlySP* score provides us with a goodness of fit of the clustering weighted by how early the last cluster starts. We score each clustering generated and then pick the smallest $k$ that achieves at least 80% of the spread between the largest and smallest *EarlySP* scores. We can set the threshold higher if want tighter clusters, at the cost of having more simulation points. This tradeoff is illustrated in the following Section IV.C.2.

#### Picking Simulation Points

After Early SimPoint picks a clustering, we determine a cutoff point in the program's execution and we consider picking simulation points only from the start of the program to this cutoff point. No intervals of execution after this cutoff point will be considered for simulation points. This bounds the amount of overhead due to fast-forwarding.

The cutoff point for a program/input is determined by first picking an early simulation point for the cluster that starts the latest in execution. The cutoff simulation point is then picked for the last cluster choosing the earliest simulation point in the cluster that has a distance within 1% of the centroid of the cluster. Once this cutoff simulation point has been determined, the simulation points for the remaining clusters are selected from all the potential intervals from the start of the program up to the cutoff interval. For a given cluster, a simulation point is chosen from these candidate intervals that is closest to the original centroid of the cluster.

### IV.C.2 Early SimPoint Results

To examine the performance of Early SimPoint we compare the results from five different algorithms. To evaluate these different algorithms, Figure IV.9 shows the number of clusters (the value chosen for $k$) that was picked for each algorithm, and this equals the number of simulation points. Figure IV.10 shows the number of instructions (in billions) it will take from the beginning of execution to reach the last simulation point. This number, minus the number of detailed instructions simulated, is the number of instructions fast-forwarded to simulate each program. Finally, Figure IV.11 shows the error rates for the different algorithms.

Different algorithms have results shown using an interval size of 1 million (1M), 10 million (10M), and 100 million (100M) instructions. Recall from Chapter III, that this interval size is the granularity in which the basic block vector (code) profile is collected and the clustering is performed. It is also the length of the simulation point when performing detailed simulation. We show a variety of different combinations of algorithm and interval size.

The 100 million interval size results show the performance when using the standard BIC SimPoint algorithm from [61] and the Early SimPoint algorithm described in this section. For both results max $K$ is set to be 10 intervals to put a limit on detailed simulation time. We provide results for the standard SimPoint algorithm using a BIC threshold of 80%. For Early SimPoint results, we use an EarlySP threshold of 100%. Setting EarlySP to have a threshold of 100% picks the clustering that achieves the highest EarlySP score. We use this for the 100M interval results, since only a small number of clusterings (max $K = 10$) are examined. With having at max 10 clusters, complex programs will be noisier within a cluster, so picking the best scoring clustering ensures that the clusters are as well formed as possible with early simulation points. The results show that the number of instructions required for fast-forwarding for Early SimPoint is 3.6 times smaller than using the original SimPoint algorithm. This comes at the cost of increasing the average error from 2.6% to 3.1%.

We also provide results for Early SimPoint using an interval size of 10 million and 1 million. For both of these results, a EarlySP threshold of 80% was used. The results for 1 million intervals shows that it has a fast-forward length 5 times longer than the 10 million interval size. This is because there are more clusters, which creates a greater chance of a cluster showing up only late in the execution. The 10 million interval size results show that they require the least amount of fast-forward time, with the fast-forward length being only 13% into the program/input run's execution on average. These results show a delicate trade-off between speed and accuracy, as well as choosing an appropriate interval size.

All of the results we have talked about to this point (Standard SimPoint and Early SimPoint algorithms) in this section are based only on the BIC, and do not use any confidence or probabilistic error bounds to guide the choosing of a clustering. We now look at using the Early SimPoint approach with the Variance SimPoint algorithm. When using EarlySP with the Variance algorithm in Section IV.B, we use the Variance algorithm for picking $k$ just as before, and then use the approach described in Section IV.C.1 to pick the simulation points. The Variance SimPoint is performed, and then simulation points are only chosen from intervals that occur from the start of execution to the simulation point of the latest starting cluster.

Figure IV.11 shows that picking early simulation points in the Variance SimPoint algorithm achieves an error rate of 3.4%, which is close to the non-early Variance SimPoint algorithm error of 2.1%. We also found that it had a very similar error bound (not shown on the figure) of 3.9% on average for 95% confidence. Figure IV.10 shows that the non-early Variance SimPoint algorithm has a fast-forward length 1.5 times longer than when picking early points for the Variance SimPoint algorithm. With 100 simulation points chosen on average for these results, the likelihood of clusters appearing only in the latter portion of execution is significant. Even with 40 samples, as is the case for the early BIC approach at 1M interval size, the ability to find all the clusters early in the execution is low.

## IV.D Summary

In this chapter we explored the primary parameters that have influence on how SimPoint and the k-means clustering algorithm behave. We described how we achieve a reasonable running time for k-means while maintaining a high accuracy by calibrating the number of clustering iterations, number and type of random seeds used, and the number of dimensions needed. We showed that its possible to sub-sample the vectors during clustering to enable the clustering of very large data sets. We also presented a binary search method over k clusterings that achieves comparable accuracy to an exhaustive search, but at a fraction of the time.

Additionally, we presented the Variance SimPoint algorithm that uses a user defined confidence and probabilistic error bound to guide the picking of $k$. This algorithm first gives priority to choosing a clustering that has well formed (based upon the code frequencies) clusters. This is to make sure that the clustering is representative across different architecture configurations. The Variance SimPoint clustering for picking samples shows that tighter probabilistic error bounds are seen when compared to random sampling when using the same amount of samples as there are simulation points. This is due to Variance SimPoint always choosing a representative sample from each cluster to make sure all the unique/different behavior in the program is captured. In comparison, for a small number of samples, random sampling tends to over/under sample some of the program's behavior resulting in looser probabilistic error bounds.

The final contribution in this chapter is the Early SimPoint algorithm whose goal is to find representative simulation points early in a program's execution to reduce fast-forward simulation time. The amount of fast-forward time required for a simulation is the number of instructions it takes to reach the last simulation point for a program/input run. When using an execution interval size of 100 million instructions, we found that Early SimPoint had a 3.6 times shorter fast-forwarded length on average than the standard SimPoint algorithm. When using an interval size of 10 million, we found the fast-forward length to be only 12% of the full execution on average, with an average CPI error of 4%. These results show that these early simulation points can be used to significantly reduce the time spent fast-forwarding to reach all of the simulation points for a program, while still providing accurate results.

## Acknowledgements

Portions of the text in this chapter are in part a reprint of the material from the paper, *Erez Perelman, Greg Hamerly, and Brad Calder, "Picking Statistically Valid and Early Simulation Points"*, in the proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2003) September 2003. New Orleans, Louisiana. The dissertation author was the primary researcher and author and the co-authors involved in the publication [52] directed, supervised, and assisted in the research which forms the basis for that material.

Section IV.A is in part a reprint of the material from the paper, *Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood, "Using Machine Learning to Guide Architecture Simulation"*, in the Journal of Machine Learning Research, 2006. The dissertation author was the secondary researcher and author and the co-authors involved in the publication [23] directed, supervised, and assisted in the research which forms the basis for that material.

# V

# Variable Length Intervals and Cross Binary Simulation Points

Architectures are usually compared by running the same workload on each architecture and comparing performance. We showed in Chapter III that SimPoint can find a small set of samples to accurately represent a program even across many different architecture configurations. Architectures can be compared by simulating their behavior on the code samples selected by SimPoint, to quickly determine which architecture has the best performance.

Architectural design space exploration becomes more difficult when different binaries must be used for the same program. There are three main scenarios we have encountered where it is necessary to compare multiple binaries during architecture simulation. In these scenarios, we are using the *same* source code for a program, producing *different* binaries from the source, and running the binaries with the *same* input. The binaries are created using different compilers and/or different optimization levels. All three scenarios involve quickly evaluating architecture design decisions, which requires representative architecture simulation.

The first area deals with ISA extensions, where a new binary is created that uses some ISA extensions, such as the 64-bit x86 extensions. In this case, we must compare the performance of the original binary, which does not use the extensions, to the performance of the new binary, which does use the extensions. For example, one of the questions Intel architects want to answer is how their new processors will perform with 32-bit (IA32) and 64-bit (Intel64) binaries, and what is the difference in performance. This requires comparing the simulated performance of two different binaries. The second case deals with examining completely different architectures, such as Itanium and 64-bit x86. In this case, different compilers will be used, and it is important to identify the same parts of execution for the simulation samples. Finally, for a new architecture, the compiler team needs to evaluate the performance effects of compiler optimizations using simulation, before working prototypes of the processor are available. In this case, a compiler may use the same ISA but produce different binaries as optimizations are enabled, disabled, and reordered.

We consider two approaches for representative simulation for multiple binaries compiled from the same source. One approach applies the standard SimPoint approach separately on each binary. SimPoint examines an execution trace and groups similar portions of execution into phases (clusters). The most representative interval from each phase is chosen as the simulation point to represent that cluster. This approach provides very accurate results when a single binary is used across different architectures, because the same simulation points are being simulated for each architecture (as was shown in Section III.E.2), and each simulation point always represents the same portion of execution.

Using SimPoint with multiple binaries for a single program can result in different clusterings for each binary. This means that part of a program's execution in one binary may be assigned to a different phase in another binary for the same program, so phases may be weighted inconsistently. More importantly, the simulation points chosen in each binary might represent different behaviors. Results in Section V.D.2 show the effects of these issues, which are especially important when determining which (binary, architecture) pair performs the best.

To address this issue, we propose a technique we call *Cross Binary SimPoint*. This approach finds simulation regions that are semantically the same across multiple binaries, and uses those regions to compare program performance. For this approach, we profile each binary with the input used for simulation, and identify a set of points in each binary that can be mapped to any other binary in the set. These *mappable points* are instructions in each binary corresponding to procedure calls and loop branches that can be consistently found in all of the binaries examined. These mappable points are potential boundaries for simulation regions. We break the execution intervals passed to SimPoint on these mappable points, and we use SimPoint to choose a set of simulation points we can map across all of the binaries. Then we use these mapped simulation points to compare performance across binaries.

One problem in picking a single set of simulation points to represent execution across multiple binaries is that a simulation point in binary $A$ may start at dynamic instruction count $X$, but the semantically equivalent part of execution in binary $B$ starts at dynamic instruction count $Y$ (and $X \neq Y$). The other problem is that the semantically-equivalent sample for binary $A$ may execute a different number of instructions than the same sample in binary $B$. Therefore, we cannot use dynamic instruction counts to identify the beginning and end of a sample. Instead, we must find samples whose boundaries correlate with source code so we can find the same sample across the execution of two different binaries.

The use of source code to define interval boundaries results in intervals that execute a variable number of instructions. For example, if we use a procedure call $C$ to define the start and end of certain intervals, it is possible that a different number of dynamic instructions will execute between call invocations of $C$. This leads us to explore a new method in how we partition program execution. Rather than break down the execution into Fixed Length Intervals (e.g. every interval spans 100 million instructions) we are faced with Variable Length Intervals (e.g. one interval spans 85 million instructions and another spans 113 million instructions, etc.). For SimPoint to handle (*VLIs*) we need to modify the clustering engine and BIC scoring algorithm to consider a dataset that can have elements with variable weights.

In this chapter we will first discuss some of the benefits of having Variable Length Intervals and then present the modifications required for enabling SimPoint to handle VLIs. We will then examine the two approaches for finding simulation points across multiple binary versions of a program and compare their accuracy for design space exploration.

## V.A Variable Length Intervals

Up to now we have concentrated on using fixed length intervals for doing phase classification. In this section we will explore some of the benefits in using variable length intervals. First we will show theoretically that fixed length intervals can result in sub-optimal phase classification. In phase classification, intervals are the building blocks for forming phases and identifying changes in phase behavior. At this level, any noise accumulated in the intervals will have ramifications on the quality of the phases detected. Fixed length intervals can result in phases that do not accurately represent the behaviors of the program. We will present real program data sets that exemplify this and motivate variable length intervals.

### V.A.1 Interval Dissonance and Harmony

To illustrate the ramifications of interval length on the representation of time-series data, we present a simple example. Figure V.1 shows what happens when a simple sinusoid is sampled at different interval durations. The top figure shows the signal, a sine wave with a constant period equal to 25.

We now consider the effects of dissonance between a fixed interval length and the period of the signal. The central figure shows what happens when we split the signal into fixed length intervals of length 11. Here we see the original signal in the background, with vertical dashed lines depicting where the intervals are split. The signal average for each interval is plotted as a point at the end of that interval, and *the solid line in this figure connects these averages to show the interval-based representation of the signal.* In this figure the solid line is very jagged, because the length of the interval is out of sync with the period length of the cyclical signal. Using fixed length intervals may require many intervals (and thus potentially result in many phases) to accurately represent the signal. We can quantify the number of intervals required to accurately represent a signal as the ratio of Least Common Multiple between the interval length and the period of the signal, and the length of the interval: $(LCM(|interval|, |signalperiod|)/|interval|)$. In this example, it would require a total of 25 intervals to accurately represent the signal using intervals of fixed length 11. If we try to cluster these intervals to understand the phase behavior of this signal, we can end up with many phases that are composed of intervals that are similar because they capture similar portions of the signal. It is not hard to see how this can be detrimental when trying to characterize the phases of real programs.

On the other hand, let us consider harmony between an interval length and the period of the signal. The bottom figure shows an interval length of 25, equal to the period of the signal. The same format is used as in the central

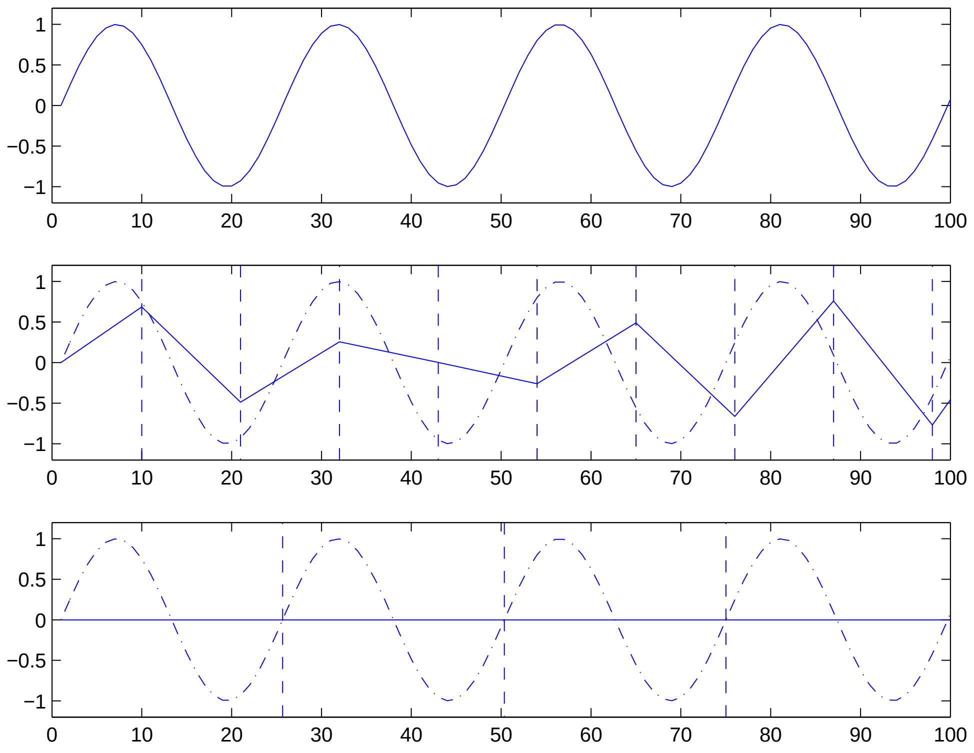


Figure V.1: An example of what happens to a signal (top figure) when it is sampled with different interval lengths. The signal in this example is a sinusoid, shown in the top figure, and the intervals it is broken into are drawn vertically in the lower two figures. The average signal for each interval is shown as the straight line within an interval. When the interval is dissonant with the period of the signal, it results in a jagged and unstable characterization as can be seen in the central figure. The optimal interval duration, shown in the bottom figure, captures exactly one cycle of the repetitive behavior, which results in a concise and stable characterization of the signal.

figure. Here we see intervals capturing entire cycles of the signal, and the resulting behavior of the intervals is constant. This is the ideal situation, since it would require exactly 1 phase to represent this signal accurately.

In this simple example, fixed intervals with length 25 can accurately represent the signal. But most programs do not exhibit simple fixed-frequency phase behavior. For example, `gzip` exhibits low-frequency phase behavior in its low-IPC phases, and high-frequency phase behavior in its high-IPC phases. It is unlikely that a single fixed interval length can accurately capture phase behavior at both these frequencies. Additionally, there are some benchmarks where the period changes over time. For example, `vpr-route` exhibits behavior patterns corresponding to each routing it tries. As it is simulated the annealing algorithm converges on a solution and it spends less and less time evaluating each solution.

### V.A.2 Program Behavior in 3D

The previous section presented a simple example where fixed length interval lengths can have significant impact on the representation of the signal. Here we examine actual program execution data, and see how it is even more susceptible to representation problems when using fixed length intervals.

To display program behavior we start with fixed length basic block vectors of the complete execution. We take this set of BBVs and reduce the number of dimensions down to three using random linear projection [61]. Then we plot each 3-dimensional vector as a point in space, and draw lines between the temporally adjacent points to show the execution order. This provides a visual portrait of how the program executes over time, through its code space. An almost constant pattern should show up as a tight cluster in space. In theory, if the boundaries between BBVs always fell perfectly on a phase transition, then we would expect to see a set of interconnected tight clusters with a lot of BBVs placed on

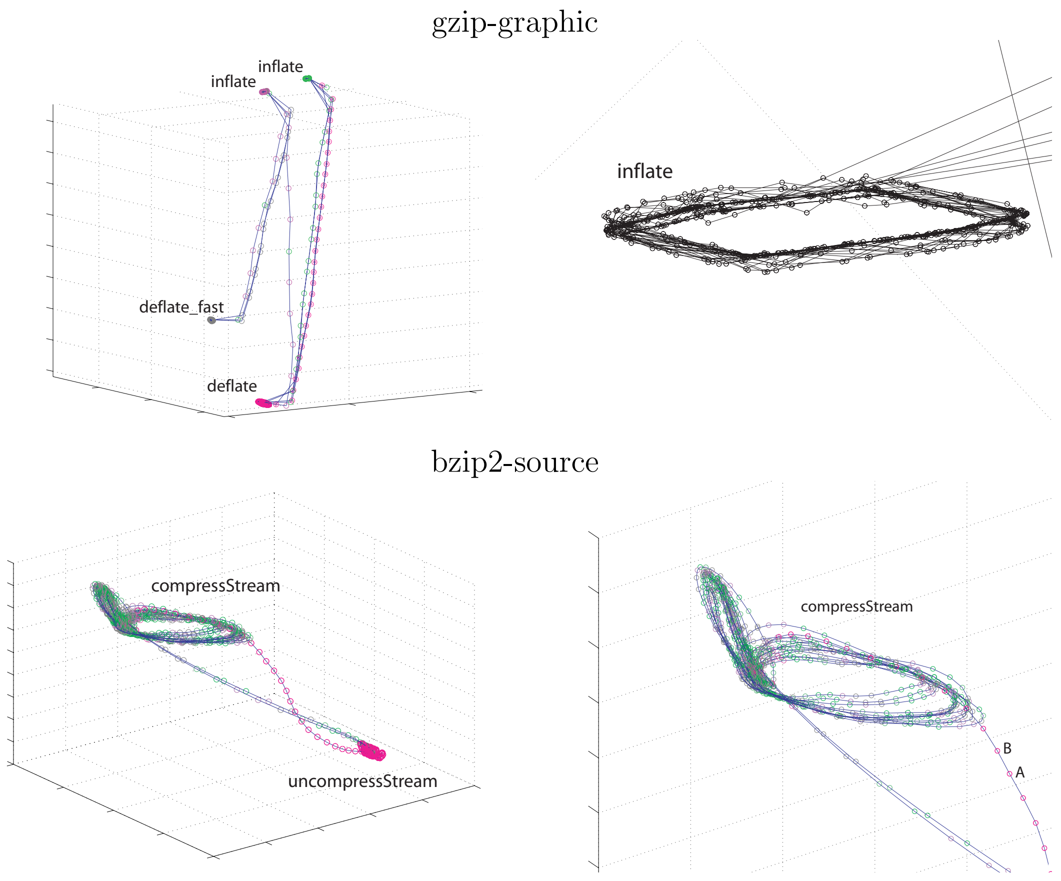


Figure V.2: The three dimensional non-accumulated representation of `gzip-graphic` and `bzip2-source`. Each point represents an interval during execution, and the line connecting the points represents the execution order in time. The right figure of `bzip2-source` has two points labeled A and B, which indicate two temporally adjacent intervals of program execution. The figure on the left plots the entire execution, while the figure on the right zooms in on a looping region in the execution. The looping structures are traversed once for each block of data compressed or decompressed.

top of each other. If the boundaries are not aligned with the periodic program behavior, then we should see *oscillations* as discussed in the prior section. The oscillations should show up as *rings* or *tori* ("donuts") if we plot these paths in 3-d space. Figure V.2 shows a 3-dimensional representation of the execution of two benchmarks: `gzip-graphic` and `bzip2-source`. A fixed length interval size of 100 million instructions was used. In both `gzip` and `bzip` there are interesting cycles that appear. For each program we zoom in on one of the cyclic regions.

In SPEC2000, the `gzip` benchmark repeatedly compress and decompresses the data a total of 5 times, at compression levels 1, 3, 5, 7, and 9. At compression levels 1 and 3, a faster version of the `deflate` algorithm is used. This time-varying program structure is clearly visible from the `gzip` graphs shown in Figure V.2. For example we see that execution bounces back and forth between `deflate_fast` and `inflate` 3 times (`deflate_fast` $\rightarrow$ `inflate` $\rightarrow$ `deflate_fast` $\rightarrow$ `inflate`), corresponding to compression and decompression at levels 1 and 3. There are 5 bounces between `deflate` and `inflate`, corresponding to compression and decompression at levels 5, 7, and 9. The vectors exhibited by each deflation and inflation phase form a torus. Each cycle around the torus corresponds to the compression or decompression of a block of data. If correctly sized variable length intervals were used, then each cycle around the region should become a single interval, instead of a series of intervals forming a torus. But because the interval length was too small, we have a large number of intervals composing this cyclical behavior. In addition, the fixed length interval is out of sync with the actual period of the phase, because different points in space are sampled on subsequent iterations around the torus.

Similarly, SPEC2000's `bzip` compresses and decompresses the data twice, at compression levels 7 and 9. Thus, execution bounces between compression and decompression three times, as seen in Figure V.2. Each iteration around the looping structures corresponds to compression of a block of data, as seen in the `gzip` plots. There are two looping structures within the compression phase - these correspond to compressing blocks with different entropy properties. `bzip` performs run-length encoding on its front end, and more time is spent in the run-length encoder on blocks with more contiguous sequences of repetitive bytes.

As seen from these two examples, the majority of a program's execution is spent in loops. The average number of instructions per loop iteration can change over time. For example, fewer instructions are typically needed to decompress a block of data than to compress a block of data. This means that the period lengths are not stable over time. An ideal fixed interval length for one section of execution (compression) may be dissonant with another portion of the program's execution (decompression). Since no single interval length will do a good job representing the program, we need variable length interval lengths that adjust to the period of the program's current behavior pattern.

### V.A.3 Supporting Variable Length Intervals in SimPoint

The SimPoint algorithm was originally designed for fixed length intervals, so modifications are required to handle variable length intervals. Here we describe the changes which allow SimPoint to handle variable length intervals. The changes come primarily in two areas: handling an increased number of intervals, and dealing with the weights associated with variable length intervals.

Different variable length intervals can represent different proportions of a program's execution, as opposed to fixed length intervals which each represent the same proportion. Each variable length interval has an associated weight we denote $w_i$, which represents the percentage of the total program execution for that interval. We have modified several parts of SimPoint so that it handles these weights.

The $k$-means clustering algorithm has two steps that it repeats: determining which cluster each interval belongs to (called the expectation step), and repositioning each cluster center to the mean of the intervals that it owns (called the maximization step). The expectation step is not changed by variable length intervals. The maximization step handles weights $w_i$ by applying them during the recomputation of the cluster centers. Cluster $j$ is computed as the weighted mean of the variable length intervals $x_i$ that belong to that cluster:

$$c_j = \frac{\sum_{i=1}^{n} w_i x_i m_{ij}}{\sum_{i=1}^{n} w_i m_{ij}}$$

Here $m_{ij} = 1$ if interval $i$ belongs to cluster $j$, and 0 otherwise, and $m_{ij}$ is determined during the expectation step. The resulting $k$-means algorithm behaves just like the $k$-means algorithm for fixed length intervals, but larger intervals have more influence than smaller intervals over the cluster center locations.

The BIC criterion that we use to choose the best clustering also needs modification to handle variable length intervals. The BIC is the log likelihood of the clustering minus a complexity penalty. We adjust the log likelihood, but keep the penalty the same. The likelihood calculation sums a contribution from each interval, so larger intervals should have greater influence. The weighted log likelihood becomes:

$$\mathcal{L} = \frac{n \sum_{i=1}^{n} w_i \log Pr(x_i)}{\sum_{i=1}^{n} w_i}$$

where $Pr(x_i)$ is the probability of interval $x_i$. In our case, this probability comes from the $k$-means clustering model of a mixture of spherical Gaussians:

$$Pr(x_i) = \frac{w_{(i)}}{\sum_{i=1}^{n} w_i} \frac{\exp(\frac{-1}{2\sigma^2}||x_i - c_{(i)}||^2)}{(2\pi\sigma^2)^{d/2}}$$

Here $c_{(i)}$ is the cluster center that $x_i$ belongs to (the center closest to $x_i$), and $w_{(i)}$ is the weight associated with cluster $c_{(i)}$ (the sum of the weights of all points belonging to the cluster; this is the denominator in the earlier equation for $c_j$).

The total weighted log likelihood function then simplifies to:

$$\mathcal{L} = \frac{n\sum_{i=1}^{n} w_i \log w_{(i)}}{\sum_{i=1}^{n} w_i} - n\log\sum_{i=1}^{n} w_i - \frac{dn}{2}\log(2\pi e\sigma^2)$$

We must also compute the variance $\sigma^2$ of the clustered intervals in a way that accounts for the weights:

$$\sigma^2 = \frac{\sum_{i=1}^{n} w_i ||x_i - c_{(i)}||^2}{d\sum_{i=1}^{n} w_i}$$

All these general equations can be simplified in the common case that $\sum_{i=1}^{n} w_i = 1$.

These changes do not change the algorithms with respect to fixed length intervals. In other words, setting all $w_i = 1/n$ (as in fixed length intervals) would produce the same results as the former SimPoint algorithm. Thus these changes allow SimPoint to smoothly handle both fixed length and variable length intervals.

## V.B Selecting Simulation Points Across Binaries

The focus of this chapter is on architecture studies we encountered that require to compare multiple binaries for the same program (no source code changes) for the same input using detailed simulation. In this section we present two approaches for representative simulation for multiple binaries compiled from the same source. The first approach uses the standard SimPoint approach for finding simulation points for each binary, which we term *Per-Binary Simulation Points*. The second approach we call *Cross Binary Simulation Points* because it finds a single set of simulation points to be used across multiple binary versions.

### V.B.1 Per-Binary Simulation Points

In this section we examine an approach that finds a separate set of simulation points for each binary being compared for a program. We apply the standard SimPoint method described in Chapter III to every binary version of a program. This results in a unique set of simulation points belonging to each binary version. By weighting and combining the simulation points for a given binary we can accurately estimate its overall performance. By estimating the performance of each binary using its unique set of simulation points we can compare the performance between the binaries. In this manner, this approach can be used to compare simulation results across binaries of the same program, but issues can arise, as described below.

SimPoint tries to capture the majority of behaviors during execution to create a faithful estimate of the complete execution, but it cannot capture every behavior. This creates bias (error). We have shown the bias is low in Chapter III, but for studies involving multiple binaries, the bias across the different binaries examined must be consistent. We have also shown that this bias (relative error) is consistent for a single binary across many different architectures in Section III.E.2, but that work only considered the case where the same simulation points are used with the same binary across different architectures.

The focus of this work is to use *different binaries* (compiled from the same source) across *different architectures*. If we choose different simulation points for each binary, then the same behaviors may not be captured in the simulation points across the different binaries. This can result in different biases used for each binary, which can cause additional inaccuracies when trying to compare the performance of many architecture/binary combinations.

Related to this is a problem of representing all the unique behaviors a program exhibits with a small number of phases (each phase having a single simulation point representative). If there are more unique behaviors than allowed phases (since an architect may limit the number of simulation points used), then SimPoint cannot represent all the behaviors as separate phases. Therefore, some unique behaviors must be grouped into the same phase. If these groupings are not performed consistently across different binaries, then simulation points from one binary will represent different combinations of unique behaviors than simulation points from another binary. Simulation results using different simulation points can not be meaningfully compared, since the different simulation points may focus on different behaviors.

Both of these issues are addressed by the next approach we examine, which finds a single set of simulation points that can be used across all of the binaries, ensuring that the same behaviors are simulated for each binary.

### V.B.2 Cross Binary Simulation Points

In this section we describe our approach for picking the same simulation points across a set of binaries for a program/input pair, and how we use these cross binary simulation points and weight them appropriately. We begin with a description at a high level of our cross binary simulation point algorithm, which has the following steps:

1. **Create Call and Branch Profile for Each Binary:** Generate a profile for each binary for the input being examined. This along with symbol information will be used to find the set of mappable points.

2. **Find a Set of Mappable Points that Exist in All Binaries:** Use symbol information, the profile counts, and source line information to find a set of instructions in the binaries that exist in all of the binaries, and serve the same purpose (procedure entry points, loop back edges, loop entry points, etc).

3. **Create Variable Length Intervals Using Mappable Points:** Use the mappable points to partition execution for one input into variable length

intervals (VLIs), where both the start and end of each VLI are mappable points. This allows us to accurately map the intervals across all of the binaries.

4. **Pick Simulation Points for the Primary Binary:** Pick a set of simulation points by running SimPoint on variable-length frequency vectors collected from one of the binaries (the “primary binary”).

5. **Map the Simulation Points to All Binaries:** Map the simulation points chosen from the primary binary to all of the other binaries, creating *cross binary simulation points* across all of the binaries that semantically represent the same part of execution.

6. **Recalculate Weights for Mapped Simulation Points:** Simulation results for each cross binary simulation point in each binary must be appropriately weighted by the size of its cluster in each binary.

We now go through each of these steps in more detail.

### Create Call and Branch Profile for Each Binary

We generate a profile for each binary for the input being examined using Pin [40], a dynamic instrumentation system. We then find instructions in the binaries that are mappable; i.e. they exist across all binaries that mark the same exact point of execution. This will be used to locate the mappable points for each binary.

For each binary we profile all procedure entry points and loop branches and keep track of the total number of times each code structure is executed. For procedures we simply keep track of how many times each procedure is executed for the entire execution. Loops on the other hand can be considered as two entities: a loop entry point and the loop body. For the loop entry points we capture how many times the loop has been entered regardless of how many iterations the loop executes each time it is entered. This provides a coarse representation of loops, similar to procedures.

We also keep track of how many times the loop body executes. This is the number of times the loop has iterated over the entire execution. Each time a loop is entered we increment the loop entry count once and increment the loop body count by the number of times the loop iterates. The loop body count provides a much more detailed picture of the loop execution and is typically much larger than the loop entry count.

We want to break down loops like this so we can use either the entry point into a loop or specific loop iteration branches as mappable points. This provides a larger set of mappable points to choose from as will become apparent in the following step.

#### Find a Set of Mappable Points that Exist in All Binaries

We use the mappable points to partition the execution of a single input into variable length intervals (VLIs), where each interval starts and ends at a mappable point. This allows us to accurately map the intervals across all of the binaries.

The mappable points consist of (a) procedure entry points and (b) loop branches in the binaries. The notion is that if we can find the exact same loop branches and procedure entry points across all of the binaries then we can use these mappable points to define interval boundaries and pick simulation points which start and end at these mappable points. These simulation points can then be mapped to any other binary in the set of binaries considered.

For all the binaries being considered for a program, we first match up the procedure entry points with the same procedure names across all of the binaries, using debug symbol information. These procedure entry points represent the same exact point in execution across all of the binaries.

We also identify matching loop branches across all the binaries. For this we use two pieces of information: execution counts from the profiles collected in the previous step, and debug line number information associated with each branch. If the execution counts and line numbers for a branch match across all binaries, then that branch represents the same part of execution across all binaries.

For both procedure entry points and loop branches, the execution count across all binary versions must match. This guarantees that the mappable points will execute the same number of times across all binaries, which allows us to specify regions in the execution of any binary in the set by using mappable points as delimiters - for example, a simulation region can start at mappable point $A$ after it has executed $X$ times and end when mappable point $B$ has executed $Y$ times. This representation allows us to capture the same regions across the executions of different binaries.

### Create Variable Length Intervals Using Mappable Points

We want to partition the execution into intervals that are close to a desired size specified by the user. The user can specify a size range (e.g. between 10 million and 100 million instructions) or a specific target size (e.g. 100 million instructions). The mappable points (markers) selected in the prior step are used to break execution into variable length intervals that attempt to satisfy the users desired interval size. Since we are dealing with program source code constructs to define interval boundaries, we do not expect to find precise fixed length intervals for a given target size; rather we expect to find intervals that are close to the target size that are also harmonic with the periodic nature of the program behavior. In this manner, we will generate variable length intervals that are both manageable for simulation time and be in sync with the program periodic behavior.

For each variable length interval we collect a basic block frequency vector, which is given to SimPoint in the following step. Only one binary (the primary binary) is profiled in this step. As the program executes, we keep track of how often each mappable marker is encountered, because any mappable marker could be used as an interval boundary. For example, if the desired interval size is 100 million instructions, and we have just executed 100 million instructions, we need to create an interval boundary on the next mappable marker we encounter (rather than force a hard boundary as is done with fixed length intervals). When the next mappable marker is reached, we record its marker ID and the number of times it has executed since the start of execution to bound the interval. We do this from the start of execution, ending intervals every time we reach the desired interval size at the next mappable marker encountered during execution. The execution count is critical, since markers can execute many times. Each (marker ID, execution count) pair uniquely identifies a specific point in execution that can be mapped to other binaries.

### Pick Simulation Points for the Primary Binary

Next we run SimPoint on the basic block vectors collected for the mappable intervals from the primary binary to pick simulation points. We use SimPoint with the modifications described in Section V.A.3, which supports variable length intervals and considers the number of instructions in each interval during the clustering process and the search for simulation points.

For the primary binary, SimPoint generates simulation points, weights for each simulation point, and phase labels for every interval. Each simulation point represents a unique phase, and the weight associated with the simulation point reflects the fraction of executed instructions in that phase.

The primary binary can be selected arbitrarily from the set of binaries available, but it should be noted that interval sizes can expand or contract depending on which binary is chosen as the primary. One interval is created approximately every 100M instructions executed by the primary binary, so if the other binaries execute more or fewer instructions between interval boundaries, the mapped intervals can be bigger or smaller in the other binaries.

The simulation points chosen from the primary binary are intrinsically mappable to the other binaries. The start and end of each simulation point is defined by a (marker ID, execution count) pair. This pair represents the simulation point across all binaries since it is based on mappable markers, and can be used during simulation to represent the start and end of that simulation point when executing each binary. These simulation points are then used across all binaries to represent the same part of execution.

### Recalculate Weights for Mapped Simulation Points

Finally, we need to appropriately weight the simulation points relative to the size of the clusters for each binary. Weights must be readjusted because the amount of execution in each phase can change across binaries.

A simulation point's weight is the fraction of the total dynamic instructions that the program executes in the phase it represents. For example, if a program executes 60% of its dynamic instructions in phase $P$, the simulation point for phase $P$ will have a weight of 60%. We calculate the correct weight for each simulation point in each binary by running each binary, and counting the number of dynamic instructions executed in each phase.

Table V.1: Memory System Configuration

| Cache Level | Capacity | Associativity | Line Size | Hit Latency | Type |
|---|---|---|---|---|---|
| FLC(L1D) | 32KB | 2-way | 64 bytes | 3 cycles | WriteBack |
| MLC(L2D) | 512KB | 8-way | 64 bytes | 14 cycles | WriteBack |
| LLC(L3D) | 1024KB | 16-way | 64 bytes | 35 cycles | WriteBack |
| DRAM | | | | 250 cycles | |

### V.B.3 Dealing With Optimized Code Regions

Compiler optimizations may modify a binary's call-loop structure. A procedure that has been inlined in the optimized version of a binary will not be mappable, since it will no longer have the procedure name and entry point associated with it. Although the process mentioned above for mapping points across the binaries does not handle this case, we have extended the mapping of points to handle some of these optimization cases.

We can detect inlined procedures by their parent nodes and the loop structure within the procedure. Consider a procedure that has a loop that executes $N$ times, which is called from another procedure that has a loop that executes $M$ times. If this procedure has been inlined and its loop structure maintained, we expect the caller to now have two loops, executing $N$ and $M$ times respectively. We can still map the loop of the inlined procedure because we can identify it based on its call count. Of course, if $N = M$, we can not determine which loop belongs to the inlined procedure based on the call counts.

## V.C Methodology

We evaluate our approach for selecting cross binary simulation points using CMP$im [29], a Pin [40] based multi-core simulator. CMP$im models an in-order processor and can simulate the performance of applications run to completion. CMP$im is configured to model a single-core processor with a three-level non-inclusive cache hierarchy with parameters as shown in Table V.1. All caches use a 64B line-size and LRU replacement policy.

To evaluate our approach, we compiled the SPEC2000 programs with debug information (-g compiler flag) on 32- bit (x86) and 64-bit (x86_64) Linux. The programs were compiled using version 9.0 of Intel's C/C++ and Fortran compilers. For each program we also compiled unoptimized and optimized versions, for a total of four binaries per SPEC program: 32-bit Optimized, 32-bit Unoptimized, 64-bit Optimized, and 64-bit Unoptimized. We then compare the performance of these binaries and examine how well the SimPoint based techniques estimate the speedup between the different binaries. We selected a subset of benchmarks from the SPEC2000 suite that would provide a representative sample and also include a wide range of programs with interesting behaviors. For each of the programs we selected we provide results using the reference inputs.

Simulation regions are represented with PinPoints files [49], which is a Pin tool chain that generates basic block vectors for each interval and then runs them through SimPoint to get the simulation points and weights. We ran each binary under CMP$im configured as above with the PinPoints file describing the simulation regions for the binary for the given input. Using statistics (reported by CMP$im) and weights (reported by SimPoint) for each simulation region, we compute a prediction for whole-program statistics and compare the results of the prediction to the actual whole-program statistics reported by CMP$im.

## V.D Results

In this section we evaluate SimPoint's performance estimates across different binaries compiled from the same program source. We show that our proposed Cross Binary SimPoint technique is an improvement over the Per-Binary

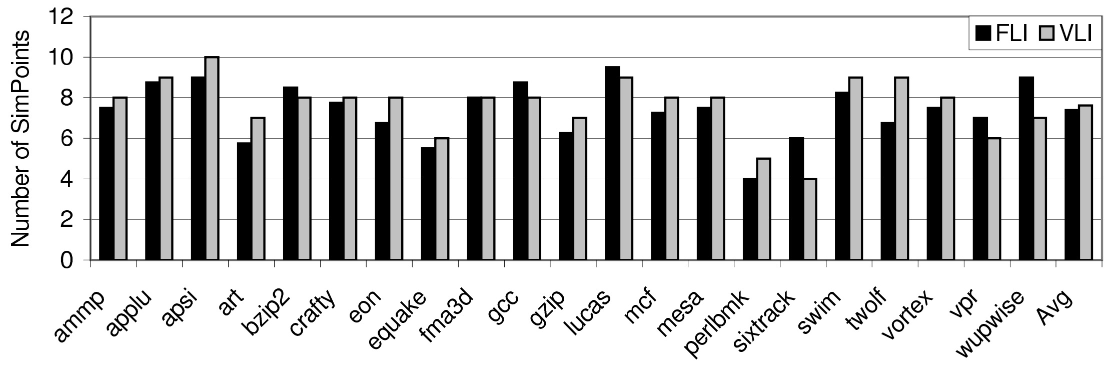


Figure V.3: Number of SimPoints for Per-Binary SimPoint (FLI) and Cross Binary SimPoint (VLI). Each bar shows the average across all four binaries.

SimPoint approach because it allows us to simulate the same regions across different binaries.

### V.D.1 SimPoint Performance Estimation

SimPoint can be run with different configurations which may result with different simulation points being selected. To fairly compare the two SimPoint methods we used the same SimPoint configurations for both techniques. We used SimPoint to find up to 10 simulation points per program since that usually achieves good accuracy in representing programs. However, SimPoint generally picks fewer simulation points than the upper limit because it usually finds a good phase characterization with fewer clusters.

Figures V.3 and V.4 show the number of simulation points picked and the average interval size respectively for each benchmark we examined. Figures V.3 shows results for Per-Binary SimPoint (FLI) and Cross Binary SimPoint (VLI), while Figure V.4 only shows results for Cross Binary SimPoint, because the interval size for Per-Binary SimPoint is fixed at 100 million instructions. We compiled four different binaries for each benchmark, and for each benchmark we

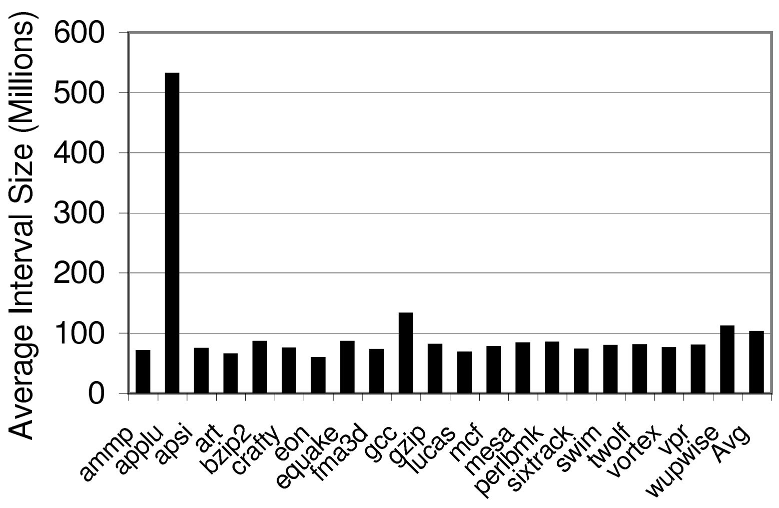


Figure V.4: Interval Size for Cross Binary SimPoint (VLI). Each bar shows the average across all four binaries. The size of each interval in Per-Binary SimPoint (using FLIs) is constant at 100 million instructions.

are showing the average across these four binaries.

Figure V.3 shows that both techniques select a similar number of simulation points on average. This is expected since the binaries all represent the same program, so we are still observing the same behaviors.

To understand the interval size differences between Per-Binary SimPoint and Cross Binary SimPoint shown in Figure V.4, recall that Per-Binary SimPoint and Cross Binary SimPoint split executions into intervals differently. Per-Binary SimPoint splits every execution of a program binary into fixed length 100 million instruction intervals, while the Cross Binary SimPoint approach produces intervals of at least 100 million instructions. In Cross Binary SimPoint, an interval ends only when a mappable marker is reached, so intervals can be larger than 100 million instructions.

In addition, Cross Binary SimPoint constructs intervals from the execution of one binary and maps the intervals to the other binaries. The same interval in another binary may not execute the same number of dynamic instructions. Suppose, for example, that an unoptimized binary executes 10 times more instructions than an optimized binary. If we use the unoptimized binary as the primary binary, we will construct mappable 100 million instruction intervals, but when the intervals are mapped to the optimized binary, the intervals will shrink to 10 million instructions on average.

This is why we see a smaller average simulation point size in Figure V.4. The intervals we constructed for the primary binary, when mapped to the other binary versions became smaller in most cases.

*applu* has a much larger interval size because our technique was unable to find mappable markers across all four binaries for large execution regions. In these execution regions, a loop calls five procedures that each solve a partial differential equation. Each of the five procedures has a similar looping structure since they are doing a similar operation. In the optimized version of this binary, all five procedures are inlined into the loop. Furthermore, the loops were split by the optimizer, and code was moved within this loop. While our technique can deal with simple cases of inlining, in this case there was not enough structure left after optimization to map the optimized code to the unoptimized code.

Figure V.5 shows the relative error in estimated CPI for each benchmark. As in the previous figure, each bar in this graph is the average across four binaries. For each binary we calculate the CPI error for that binary using the simulation points compared to a full simulation of the program. We then averaged this CPI error across the four binaries for the results shown. On average we see that both techniques accurately estimate the performance of the programs. The per-binary technique accurately estimate the performance for each binary when compared to the full execution of that binary, using a different set of simulation points for each binary. Figure V.5 also shows that cross binary simulation points

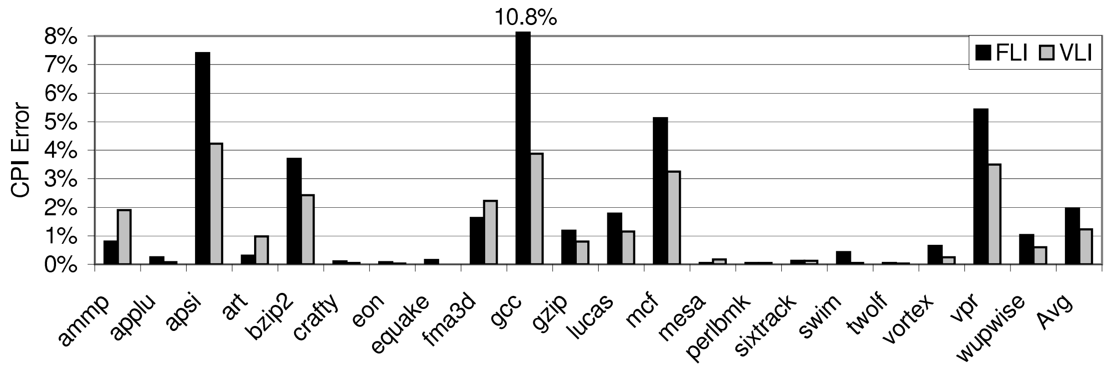


Figure V.5: CPI Error for Per-Binary SimPoint (FLI) and Cross Binary SimPoint (VLI). Each bar shows the average across all four binaries.

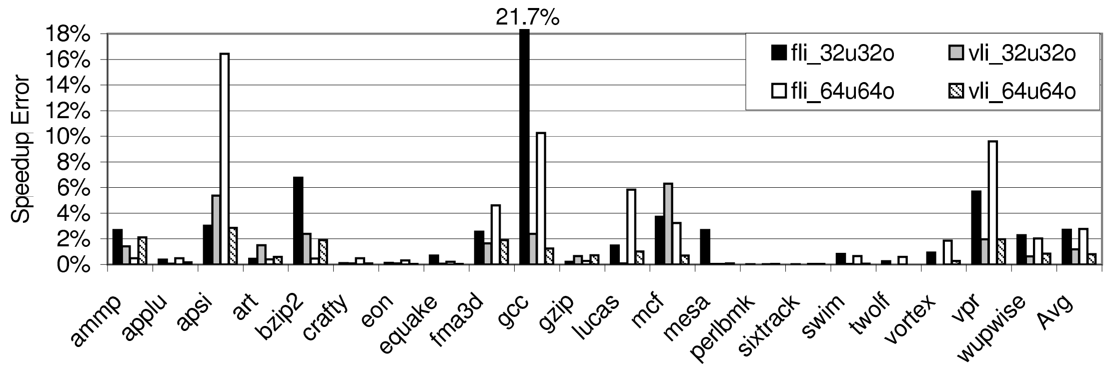


Figure V.6: Speedup error for Per-Binary SimPoint (fli) and Cross Binary SimPoint (vli). Speedup is computed across different binary pair configurations on the same platform and the error is based on how closely the estimated speedup is to the true speedup. 32U is 32-bit Unoptimized, 32O is 32-bit Optimized, etc.

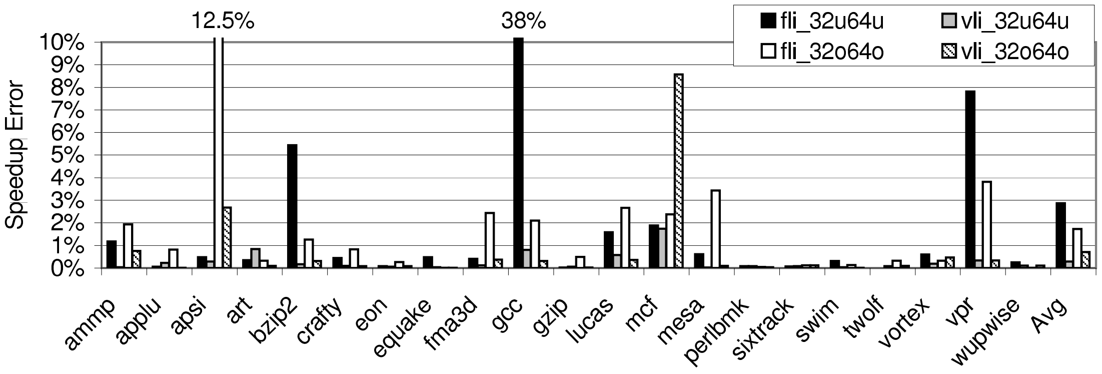


Figure V.7: Speedup error for Per-Binary SimPoint (fli) and Cross Binary SimPoint (vli). Speedup is computed across different binary pair configurations across different platforms (32-bit and 64-bit) and the error is based on how closely the estimated speedup is to the true speedup. 32U is 32-bit Unoptimized, 32O is 32-bit Optimized, etc.

achieve accurate performance estimates, but the figure does not show that the biases are consistent in the errors across the binaries, which is the focus of the next result.

### V.D.2 Speedup Comparison

When using sampled simulation for design space exploration, it is very important to have a consistent bias across the experiments to make meaningful performance comparisons. Here we calculate the actual speedup between different binaries and compare how the two SimPoint methods perform in estimating the speedup. We find that the Cross Binary SimPoint method has a more consistent bias than the Per-Binary SimPoint method, and is more accurate when comparing results across different binaries.

Figures V.6 and V.7 show the error in speedup estimation across several binary pair configurations. Figure V.6 shows binary pair configurations on the same platform varying the optimization levels, while Figure V.7 shows binary pair configurations across platforms for the same optimization level. Each Figure shows how closely the estimated speedup of either Per-Binary or Cross Binary SimPoint is to the true speedup. We compute the error in speedup as the following: $|(TrueSpeedup - EstimatedSpeedup)/TrueSpeedup|$. The $TrueSpeedup$ is computed as the ratio of total cycles executed for two binary versions. For example, the $TrueSpeedup$ for $32u32o$ configuration is the ratio of the number of cycles executed with the 32-bit unoptimized version and the 32-bit optimized version. The $EstimatedSpeedup$ is computed just like the $TrueSpeedup$ but instead of using the true number of cycles that execute we are estimating the number of cycles using sampled simulation. The error in speedup tells us how close our speedup estimates are to the actual speedup seen between the two binaries.

For each benchmark in these figures we show 4 pairs of configurations for speedup analysis: 32-bit unoptimized to 32-bit optimized and 64-bit unoptimized to 64-bit optimized in Figure V.6 and 32-bit unoptimized to 64-bit unoptimized and 32-bit optimized to 64-bit optimized in Figure V.7. For each configuration we estimate the speedups using the Per-Binary and Cross Binary SimPoint methods and compute the speedup error as described above.

The results in these figures show that Cross Binary SimPoint results in a lower error in speedup estimation on average than Per-Binary SimPoint, for the binaries we examine. This result can be explained by the lack of behavioral consistency in samples that are chosen as simulation points across the different binaries for the per-binary approach. Whenever we use simulation on a small portion of a program to estimate the performance of the whole program, there will be some unavoidable error for behaviors that are not well-represented in the simulation. We call this a bias in the simulation, and it occurs because it is impossible to simulate a behavior that is not represented by a simulation point.

Table V.2: Phase comparison across 32-bit unoptimized and 64-bit unoptimized `gcc` binary versions

| | `gcc/166` | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 32-bit Unoptimized | | | | | 64-bit Unoptimized | | | | |
| | Phase | Weight | True CPI | SP CPI | CPI Error | Phase | Weight | True CPI | SP CPI | CPI Error |
| VLI | 1 | 0.35 | 3.16 | 3.15 | 0.2% | 1 | 0.28 | 2.97 | 2.97 | -0.1% |
| | 2 | 0.26 | 3.99 | 2.93 | 27% | 2 | 0.21 | 4.11 | 2.93 | 29% |
| | 3 | 0.14 | 4.47 | 5.17 | -16% | 3 | 0.17 | 5.49 | 6.34 | -16% |
| FLI | 1 | 0.36 | 3.16 | 3.16 | 0% | 1 | 0.22 | 2.98 | 2.97 | 0.5% |
| | 2 | 0.31 | 6.54 | 2.90 | 56% | 2 | 0.18 | 6.04 | 7.04 | -17% |
| | 3 | 0.09 | 5.00 | 4.04 | 19% | 3 | 0.16 | 6.66 | 7.19 | -8.0% |

Because Cross Binary SimPoint uses the same execution regions across different binaries, errors in performance estimation due to lack of representation will occur consistently across all the binary executions. Thus the error that occurs due to bias is consistent across all our estimates. This consistency allows us to obtain performance estimates that are more accurate when comparing performance across binaries, allowing us to make better design decisions.

### Phase Bias Comparisons

In the Per-Binary SimPoint approach we are picking a different set of simulation points for each binary version. Each set of simulation points will be accurate in representing the overall execution for that binary. However, a particular program behavior may be more representative in one binary (since a simulation point may be chosen directly from that behavior), and less representative in another binary. Thus the unavoidable error due to using a small fraction of program execution to represent the whole program execution will not be consistent across all the binary versions using the Per-Binary SimPoint approach. Per-Binary SimPoint can give semantically similar simulation points that represent common program behaviors across binaries, but it is not guaranteed.

Table V.3: Phase comparison across 32-bit optimized and 64-bit optimized apsi binary versions

| | apsi/ref | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 32-bit Optimized | | | | | 64-bit Optimized | | | | |
| | Phase | Weight | True CPI | SP CPI | CPI Error | Phase | Weight | True CPI | SP CPI | CPI Error |
| VLI | 1 | 0.52 | 3.04 | 2.91 | 4.5% | 1 | 0.52 | 2.59 | 2.44 | 5.9% |
| | 2 | 0.19 | 3.57 | 3.10 | 13% | 2 | 0.18 | 3.16 | 2.66 | 16% |
| | 3 | 0.05 | 4.66 | 4.70 | -0.9% | 3 | 0.05 | 3.64 | 3.63 | 0.3% |
| FLI | 1 | 0.71 | 3.50 | 3.00 | 14% | 1 | 0.65 | 2.77 | 2.50 | 0.9% |
| | 2 | 0.05 | 4.58 | 4.61 | -0.7% | 2 | 0.08 | 5.34 | 3.39 | 37% |
| | 3 | 0.05 | 4.60 | 4.63 | -0.7% | 3 | 0.06 | 7.61 | 7.55 | 0.8% |

As a particular example of the benefits of consistent bias for making design decisions, we consider two benchmarks in detail: `gcc` and `apsi`. Both benchmarks have higher speedup error using Per-Binary SimPoint than our Cross Binary SimPoint technique. Tables V.D.2 and V.D.2 compare phase statistics across two binary versions for `gcc` and `apsi` respectively. Table V.D.2 compares the largest three phases found with Per-Binary SimPoint and Cross Binary SimPoint across 32-bit unoptimized and 64-bit unoptimized `gcc` binary versions. Table V.D.2 compares the largest three phases found with Per-Binary SimPoint and Cross Binary SimPoint across 32-bit optimized and 64-bit optimized `apsi` binary versions. Both tables show for each phase the phase ID, the weight of the phase (the percentage of executed instructions in that phase), the true CPI of the phase (the average CPI across all intervals in that phase), the estimated CPI using the SimPoint techniques, and the relative error between the true CPI and the SimPoint CPI.

Table V.D.2 shows the problem of picking simulation points for each binary using the per-binary (FLI) approach. For `gcc` we see that the weights for the three phases for FLI changes from 36% to 22% for phase 1, then from 31% to 18% for phase 2, and then from 9% to 16% for phase 3. This shows that for the 64-bit binary, a large portion of execution is grouped into different phases compared to the 32-bit binary. This is further shown by the significant changes in CPI error for the phases between the two binaries. These weights and CPI errors are the bias that SimPoint introduces. This bias is perfectly fine to have when using a single binary to compare options across a design exploration, because the bias is consistent and does not change.

When different binaries are used to explore a design space, the biases can change, but they must be consistent. Table V.D.2 shows that per-phase biases can change significantly between the binaries when using the Per-Binary (FLI) SimPoint approach. For example, when using FLI the second phase in `gcc` has an error of 56% for the 32-bit binary and -17% for the 64-bit binary. Similarly the 32-bit binary has 19% error and the 64-bit binary has -8% error. This change in bias is the reason for the 38% error in speedup for `gcc` in Figure V.7.

In comparison, the Cross Binary (VLI) SimPoint approach proposed in this chapter has a consistent bias across the phases. This is because the simulation points chosen across the binaries represent the same part of execution. The weights have slightly changed for VLI, but this is to be expected due to differences in compilation. Similar results can be seen for `apsi` in Table V.D.2. For `apsi` the bias for the per-binary FLI approach for phase 2 changes from -0.7% to 37%, whereas the bias is kept consistent across the phases for our Cross Binary SimPoint approach.

Finally, we want to emphasize that the error seen for SimPoint for a given phase in Tables V.D.2 and V.D.2 is to be expected. The error can occur because the single simulation point used for the phase did not represent all of the behavior in that phase, just the majority of the behavior for the phase. SimPoint is used to find a small set of the most representative behaviors, and because of this not every behavior can be appropriately represented. From our several years of using SimPoint, our experience has shown that the majority of behaviors will be captured, and this allows us to perform accurate architecture design comparisons. This is achievable when using a single binary for design exploration, since the same simulation points are used, which results in a consistent bias and error across the architectures examined. When using multiple binaries, our Cross Binary SimPoint is needed in order to capture the same representative part of execution for the simulation points across the different binaries. This maintains a consistent bias and error for the cross binary simulation points.

## V.E Related Work

We now briefly compare our approach to a prior technique in simulation that use procedure and call boundaries to define intervals of program execution.

Huang et al. [39] considered procedures and loops to partition a program's execution. The partitioning determined where and when statistical samples should be taken during architecture simulation. Their analysis broke up a program's execution at static call sites, and if a procedure executed for too long, they divided the procedure's execution into its major loops. To determine the sample rate, they examine the variability of several architecture metrics for each program region.

Our approach focuses on a simulation scenario not addressed in the above research. The above research examined only applying their approach to one binary for a single program/input combination, whereas we are focusing on how to effectively compare multiple binaries for a single program/input combination. The difference in our approach is that we need to (a) perform analysis to find mappable points across all of the binaries, even in the face of compiler optimizations, whereas the prior techniques break intervals at any arbitrary branch point, and (b) we have to correctly calculate the weights of the mapped simulation points for each binary, which the prior techniques did not have to deal with.

## V.F Summary

In this chapter we explored variable length intervals and presented a new simulation technique to compare multiple binaries of a program.

First, we showed using real program datasets that the majority of a program's execution is spent in loops. The average number of instructions per loop iteration can change over time. This means that the period lengths are not stable over time. An ideal fixed interval length for one section of execution may be dissonant with another portion of the program's execution. Since no single interval length will do a good job representing the program, we presented the concept of variable length interval lengths that can capture the varying period of the program's behavior pattern. We also modified SimPoint to consume variable length intervals, which is essential for our Cross Binary Simulation Points.

Researchers testing a new ISA extension, examining a new architecture, or trying a new compiler optimization may need to analyze and evaluate performance across different binaries of a program. Due to the slow nature of performance simulators, it has become a standard practice to use representative sampling simulation techniques. We examined two approaches for simulation when there are multiple binaries for a single program/input.

The first approach simply applies the existing SimPoint approach separately on each binary, creating a different set of simulation points for each binary. This approach can accurately estimate the performance for each binary by using different simulation points for each binary, but the approach can have significant error when comparing performance across binaries, since the different simulation points may emphasize different behaviors.

The second approach identifies simulation points that represent the same behaviors across all binary representations of a program. This allows us to simulate the same parts of execution as we change the ISA or compiler optimizations during design space exploration. Our approach finds phase transitions during execution that are identifiable in all of the binaries considered. We use these phase markers along with SimPoint to pick simulation points to represent the full execution of the program, and to identify the exact same start and end of execution for the simulation points in each binary. Our results show that this method does not suffer from changing biases that can occur with the first approach, so cross-binary simulation points can be used to accurately compare performance across binaries.

## Acknowledgements

The text in this chapter is in part a reprint of the material from the paper, *Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, and Brad Calder, "Cross Binary Simulation Points"*, in the proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS 2007) April 2007, San Jose, California. The dissertation author was the primary researcher and author and the co-authors involved in the publication [54] directed, supervised, and assisted in the research which forms the basis for that material.

The text of Section V.A and Section V.A.3 are in part a reprint of the material from the paper, *Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder, "Motivation for Variable Length Intervals and Hierarchical Phase Behavior"*, in the proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2005) March 2005, Austin, Texas. The dissertation author was the secondary researcher and author and the co-authors involved in the publication [33] directed, supervised, and assisted in the research which forms the basis for that material.

# VI

# Phase Analysis on Parallel Applications

In this chapter we examine applying the phase analysis algorithms and how to adapt them to parallel applications running on shared memory processors. Our approach relies on a separate representation of each thread's activity. We first focus on showing its ability to identify similar intervals of execution across threads for a single run. We then show that it is effective at identifying similar behavior of a program when the number of threads is varied between runs. This can be used by developers to examine how different phases scale across different number of threads. Finally, we examine using the phase analysis to pick simulation points to guide multi-threaded simulation.

## VI.A Methodology and Metrics

The data presented in this chapter was collected on a 4-node Itanium II. This platform has four Intel $^{TM}$Itanium II processors at 1 GHz with a 256KB L2 cache and a 3MB L3 cache, an 870 Intel $^{TM}$chipset, and a 400MHz front side bus. This is a multi-processor system, and each processor has its own L2 and L3 cache, and only main memory is shared across the processors. For this work we only run one thread per processor node.

Our data collection methodology utilizes the commercially available Intel VTune Performance Analyzer [72]. Our phase analysis framework processes the VTune output file collected from the execution of a program on any platform for which VTune is available.

The applications examined in this work are a selection from an extensive set of experiments we ran. We experimented with both OpenMP and p-thread, and on task-parallel applications. We found the definition of phases proved to be coherent even when different threads clearly do not execute the same phase at the same time. To show the range of parallel phase behavior found we provide results for the OpenMP C version of the NAS parallel Benchmarks (NPB) [44], which are derived from computational fluid dynamics (CFD) applications, consisting of five kernels and three pseudo-applications, and two more benchmarks OpenMP-parallelized:

- SNP (Single Nucleotide Polymorphism) is capable of detecting structure around a single nucleotide polymorphism in a DNA chain. This application has been coded by using Intel's Open Source Probabilistic Network Library (PNL) [67]
- SVM RFE (Support Vector Machine Recursive Feature Elimination [22]) is based on the state of the art Support Vector Machine classification algorithm and is used for eliminating gene redundancy in micro-array data analysis.

Both of these applications repeatedly access very large databases, and apply general purpose machine learning and data mining algorithms to bioinformatics applications.

### VI.A.1 Metrics for Evaluating Phase Classification

Phase detection is performed in a completely performance-independent fashion, solely based on code signatures as described in Section III.B.1. A major assumption underlying this and most of the phase analysis related work is that similar code execution intervals yield similar performance. To this end, we decide to also collect performance data at runtime using VTune while collecting the code signature. This allows us to later verify the assumption and validate our work.

The metrics we examined from VTune are CPI, together with L2 and L3 Hits and Hit-Rates. We have found that these are the key metrics to analyze in order to understand the performance of a multi-threaded application, which is often bounded by issues such as data-reuse and conflicts between caches.

We measure the effectiveness of our phase classifications by examining the similarity of program metrics within each phase. After classifying a program's intervals into phases, we compute the phase based standard deviation for each metric (e.g., CPI, data cache hit rates). This is computed by combining the weighted standard deviation from each phase. We weight each phase's standard deviation by the relative size the phase represents from the entire execution, since phases can have significantly different execution spans. We compare this phase-based standard deviation to the standard deviation seen when looking at all of the intervals of the program's execution. Better phase classifications will exhibit lower per-phase standard deviation for an architecture metric when compared to the standard deviation of the complete execution. For example, if all of the intervals in the same phase have exactly the same CPI, then the per-phase standard deviation will be zero.

## VI.B Profiling Program Behavior

In this section we provide a description of the basic structure we use in order to represent code execution in a given interval.

### VI.B.1 Extended Instruction Pointers

Davies et al. [15] proposed using hardware sampling of instruction pointers to represent code signatures for finding phase behavior, and such an approach was also used by Annavaram et al. [2] to try to find phase behavior in database workloads.

In this work we also use Extended Instruction Pointers (EIPs) to find phase behavior. An EIP is the memory address of an instruction, analogous to PC. The EIPs are extracted while running an application on native hardware. VTune, a commercially available software performance analyzer for Intel$^{\text{TM}}$ architectures [72] is used to collect the EIPs. We focus on this approach, instead of instrumentation, since VTune has the ability to non-intrusively analyze any application running on native hardware with negligible overhead. The underlying VTune driver monitors a large number of performance/code execution attributes stored in the embedded event counters of the Intel processors while a program is being executed on real hardware. It collects information, such as EIPs and CPI, which are then used to perform code clustering, phase analysis, and validation.

VTune interrupts execution at regular intervals of instructions executed and records the EIP and event counter totals (e.g. clock tick count, instruction count.) Sampling at a high frequency can significantly increase execution overhead. Conversely, too low a sampling frequency will lead to sparse data that could compromise phase analysis. Based on our experimental data, we set the VTune sampling rate to be once every hundred thousand instructions. It proved to be a good trade-off between execution overhead and collecting adequate sampling data. At this sampling rate, the typical overhead of using VTune is not more than 2%.

Annavaram et al. [2] used the EIPs to create an *Extended Instruction Pointer Vector* (EIPV), which is a one dimensional array where each element in the array corresponds to one Extended Instruction Pointer in the program execution (similar to a Basic Block Vector as described in Chapter III). The EIPV contains all zeroes at the beginning of each interval of execution. During each interval, the number of times each EIP occurs during sampling with VTune is recorded, and each EIP's final interval count is stored in the EIPV.

The intuition behind this is that the behavior of the program at a given time is directly related to the code executed during that interval as was discussed in Chapter III. The EIP vectors can be used as code signatures for each interval of execution: each vector tells us which portions of code are executed, and how frequently. For a suitably chosen sampling frequency, the sampled information gives a sufficiently accurate estimate of the frequency of execution of significant EIPs within a given interval. By comparing the EIPVs of two intervals, we can evaluate the similarity of those two intervals. If the distance between the EIPVs is small, then the two intervals spend about the same amount of time in roughly the same code, and therefore the performance of those two intervals should be similar.

### VI.B.2 Sampled Basic Block Vectors

The construction of EIPVs has been proposed in [15] as an alternative to Basic Block Vectors (BBVs). Because we are sampling the execution we are capturing only a small fraction of the EIPs that execute. This is necessary to maintain low overhead with VTune and not bias the performance of the application we are monitoring. This means that the EIPVs can be very sparse and this can generate sampling noise similarity measurements. For example, if we have two intervals that execute the same code regions but we are sampling different EIPs in each interval, then the EIPVs for those intervals will not be considered similar.

To address this issue we present a method to map EIPs to basic blocks in order to reduce some of the sampling noise [35]. From the definition of a basic block we know that if we execute one instruction in the basic block we will execute all instructions in it. The mapping of EIPs to basic blocks means that if we sampled different EIPs in the same basic block, both will show up as the same dimension. This helps reduce the artificial noise between two EIPV code signatures that are classified as different because there are different counts between the two vectors for EIPs from the same basic block.

In this chapter we map the EIPs down to basic blocks to create sampled basic block vectors. We implemented mapping EIPs to basic blocks using the Itanium version of Pin. We statically process a binary, marking every instruction as a conditional branch, a conditional branch target, both, or neither, and then use these markings as boundaries in assigning a block ID to every instruction in the binary. We then use this ID mapping (a dimension in the sampled BBV) to coalesce all dimensions in the EIPV that map to the same block-ID into a single dimension with weight equal to the summation of the weights of the remapped EIP dimensions, producing a sampled BBV. For the results in this chapter we use the sampled BBVs to find the phase behavior in parallel programs.

## VI.C Discovering Phases for a Single Parallel Run

Parallel applications can have multiple threads executing different parts of the binary at the same time. This presents new challenges in program characterization and phase analysis. In this section we provide a detailed description of the algorithm we use that characterizes parallel applications. In this section we focus on analyzing a single parallel run, and in Section VI.D we describe how to extend this to examine behavior varying the number of threads.

### VI.C.1 Phase Analysis Merging All Threads Together

The foundation of parallel application characterization relies on preserving the parallel structure of execution during phase analysis. The thread level behavior of the application is the framework through which the parallel structure is perceived. Hence, the representation of the thread level behavior is a critical component in the analysis.

One possible thread representation is an agglomerated (combined) view of all the activity across the different threads to create intervals of execution. This is achieved by agglomerating the execution samples collected from individual threads into a single execution trace and creating fixed length intervals from that. For this approach the different thread behaviors would be intermingled into one trace as they occur during execution. This trace can be run through existing phase analysis techniques, but has the following drawback: the phases discovered in this trace do not apply to any individual threads, but instead apply to the combination of behaviors from all threads. This provides the following issues: (1) it is difficult to interpret the behavior of individual threads and (2) it is hard to validate the behavior of an agglomerated interval, since it represents a combination of parts of several threads of execution. If threads execute at different rates relative to each other, then the phase representation will not be consistent across the intervals formed. We tried the agglomerated method, and found it does not provide enough per thread information to understand the parallel phase behavior.

### VI.C.2 Keeping the Thread Data Separate

Instead of agglomerating the behavior of the threads, we found that representing each individual thread in the application independently is the key to parallel phase analysis. For this approach each thread has its own set of sampled BBVs for its execution. In this manner each thread is an independent entity. We do this, because we want to find phase similarities across threads.

### VI.C.3 General Algorithm

The parallel phase analysis algorithm is similar to the standard SimPoint algorithm at a high level (Chapter III). The execution of the application is broken down into intervals, which are then clustered into a set of phases. It is different, however, in how it handles the data from multiple threads at different stages in the algorithm. Several modifications are essential to ensure that the phases found between the threads are consistent and can be compared across the threads. The algorithm is described in the following steps:

1. We first collect code execution frequencies of EIPs for each thread in the application. This data is partitioned into intervals of 100 million instructions, where each thread has a unique set of intervals representing its own execution.

2. For each thread we have a trace of intervals that represents its execution. We then generate a large combined trace for all the threads by concatenating the interval traces from each thread. This step does not contaminate the per thread execution behavior, since each thread occupies a non-overlapping sub-section in the trace. The purpose of this step is to find intervals of execution that are similar across the different threads.

3. We then cluster all the intervals into a set of phases. For this step we use

the $k$-means [41] algorithm of SimPoint. SimPoint determines the number of phases by clustering over a range of values, and then use the Bayesian Information Criterion (BIC) [50] to quantify the goodness of each clustering. In this work we considered a range of up to 10 phases, and a larger range can be used for attaining higher accuracy.

This algorithm finds the phases in a parallel execution. Each phase defines a particular code behavior in the parallel execution that is independent of the multi-thread interaction, since we are only looking at the code signatures for the intervals on a per thread basis.

We also find similar execution across threads, since we cluster all of the threads' intervals together at the same time. The vectors are formed on a per-thread basis, but the clustering is performed looking at all of the intervals from all of the threads at once. Therefore, the phases discovered are applicable across all threads, where similar behavior observed across multiple threads will be captured and characterized as one behavior. A method similar to this was independently developed and briefly examined in [49].

### VI.C.4 Thread Execution Reconstruction

The phase analysis described above discovers the phase behavior across threads by clustering the BBVs collected in all thread executions in terms of instruction count. We now describe how to map the phase information found to the threaded program's execution over time. In a parallel execution there may be synchronization points, where some threads are waiting for other threads before continuing execution or certain threads are spawned in the middle of execution. We take this into consideration when forming the fixed length interval, so that an interval does not span across these types of stalls. In addition, we need to take this into consideration when mapping the phase classification back to a parallel execution trace.

The goal here is to identify visually for a user what phase of execution each part of a thread's execution is in. VTune outputs EIP samples in the order they were collected across the multiple processors and threads. This provides a complete sequential ordering and EIP trace among the threads of execution. We use this trace to reconstruct a total count of instructions retired (global instruction count) across all threads of execution. This allows us to correlate when a fixed length interval, which was assigned to a phase for a thread, occurs during execution relative to the intervals from other threads. We can then examine cross-thread phase behavior over time, since intervals from different threads can be grouped into the same phase. Note, since the intervals were formed using only per-thread instruction counts, the start and end of the intervals may not be synchronized across all threads. This leads us to the step of reconstructing exactly when each interval starts and stops in terms of the overall execution time.

Figure VI.1 graphically shows the phase classification for the NAS benchmark `ft.B` when using 4 threads with respect to instructions retired over time. The top most sub-figure shows the phases color coded across the entire execution. In this plot, the $x$-axis shows the total number of instructions (global instruction count) retired for all the threads. The global instruction count, which is across all threads being executed, was also gathered along with the EIPs during VTune sampling. The global instruction count was not used to perform the phase classification, we just use it to map the per-thread phase intervals to execution time (represented by global instruction count). The $y$-axis is partitioned into 4 sections, 1 per thread. Each phase has a particular color (or shade). If a thread has an interval of white it means that no instructions were retired during that interval. The lower sub-figures are showing L3 cache references, and L3 cache hit rates. The $x$-axis in these sub-figures are showing the number of instructions

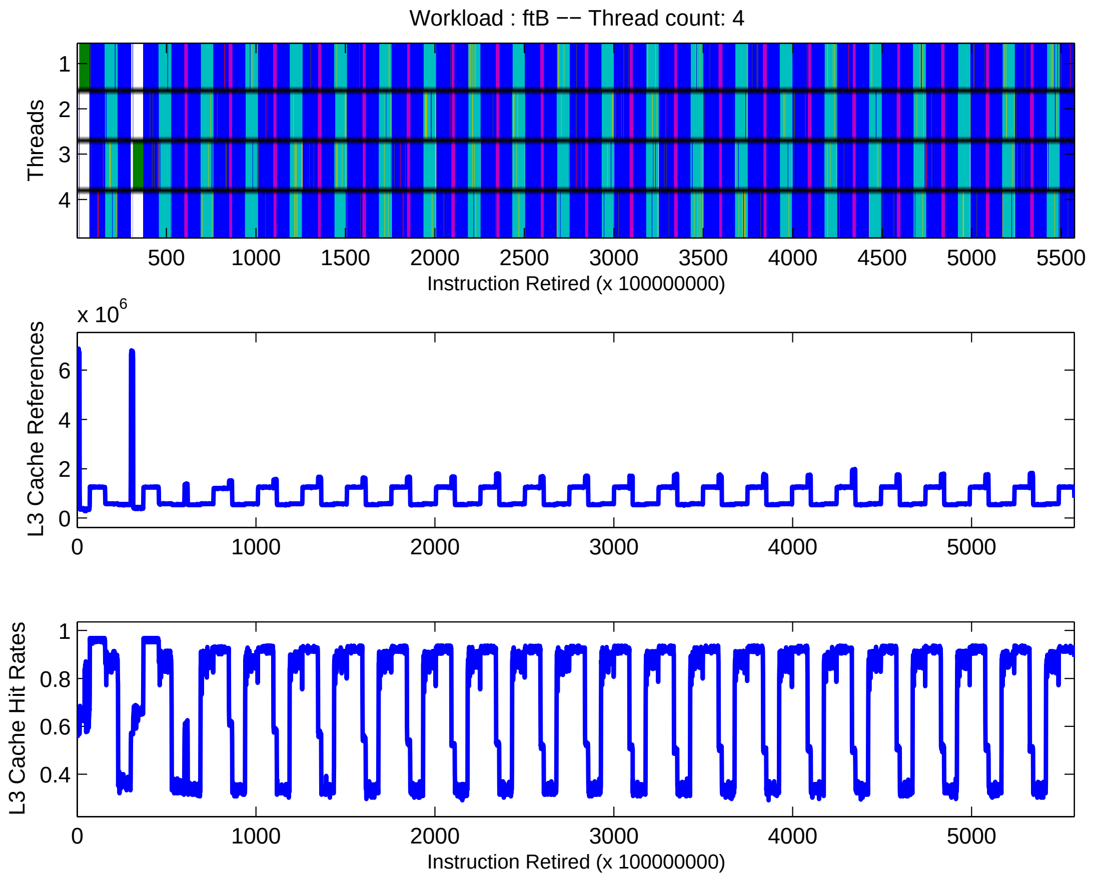


Figure VI.1: Phase classifications and L3 performance metrics for a four-threaded run of ft.B. Phase classifications are applied to each thread independently.

retired, and are equivalent to the $x$-axis in the top sub-figure for phases.

When mapping a per-thread's intervals to the global instruction count, if there is a large gap (greater than an interval size) in a per-thread's execution with respect to the global instruction count, then a blank (white) interval is shown representing that the thread was stalled or context switched out during that part of execution. It is interesting to see how the phase analysis, performed ignoring any time or similarity information among threads, does indeed automatically detect phases coherently with the execution flow across threads. The benchmark `ft.B` represented in Figure VI.1 is a data-parallel application, and the phase analysis correctly places all threads in the same phases at the same time for the majority of the execution. However, occasionally different roles for the different threads are also seen; this occurs in the figure whenever one thread is in charge of initialization or collecting results. When this occurred the code signature formed clearly identified that execution as different.

White intervals visible in the picture represent intervals of execution where one or more threads are stalled because a portion of the code is not parallelized or requires a smaller number of threads than the available thread-count. It can also happen for other synchronization issues, or OS activity; note that this naturally happens at the beginning or end of a short serial phase.

### VI.C.5 Single Parallel Run Results

We now examine the performance of our parallel phase analysis on the NAS benchmark suite and two data mining benchmarks. The programs and methodology used are described in Section V.C.

#### Reduction of Variance

The goal of parallel phase analysis is to group together program execution across the different threads by only looking at code signatures. If the phase classification worked well, then the variance in CPI, L2 and L3 cache hit rates should decrease between all of the intervals within a phase when compared to the variance seen across the complete execution of the program. For all of the results an interval size of 100 million instructions and a maximum limit of 10 phases (clusters) was used when performing the phase analysis.

Table VI.1 shows the mean and standard deviation for CPI (cycles per instruction), and L2 and L3 caches hit rates *for the full execution.* The number in parenthesis is the standard deviation. Results are shown for each program for a 2-thread parallel run and a 4-thread run. The last column is the number of phases chosen by our analysis for that run. Note, all of the CPI and hit rate results are the average and standard deviation seen across all of the intervals of execution. For example, `svm` 2-thread has a CPI of 0.87 with a standard deviation of 0.32 over all of the intervals of execution, its L3 hit rate is 48% with a standard deviation of 21%, and its execution was clustered into 6 phases.

The NAS benchmark suite shows several different potential program behaviors, where `ep.B` and `is.B` are at the two extremes. In Table VI.1, the program `is.B` has a huge standard deviation in CPI (+/- 10) for 4-threads because there are intervals of execution that have a spike greater than 10 CPI. The serial part of the code for `is.B` covers approximately 80% of the execution (instructions retired), and this makes it a very peculiar one among the NAS benchmarks. In comparison, the results for `ep.B` show that across all of execution there is a low standard deviation across all of the metrics.

Table VI.1 shows that the CPI can be stable or increase when going from 2-threads to 4-threads. This happens whenever the speedup obtained by executing the benchmark in parallel does not scale linearly with the thread count, and the parallelization introduces overhead. The CPI shown is calculated on a per-thread basis, and does not represent a measure of speed of execution of the overall program on the machine, but rather a measure of speed of execution of instructions on each single thread.

We now examine how well the phase classification, based on code, worked in terms of the underlying architecture metrics. If the phase analysis groups the intervals correctly, then we should see reductions in the standard deviation of these architecture metrics, when examining them across all of the intervals within a phase. Figures VI.2, VI.3 and VI.4 show the reduction in the standard deviation of CPI, L2 and L3 hit rates over the baseline values shown in Table VI.1 for the

Table VI.1: Full execution CPI, L2 and L3 hit rates with standard deviation across all intervals of execution for each benchmark with 2 and 4 threads (#T). The number of phases (#P) is equivalent to the number of simulation points.

| BM | #T | CPI | L2 HR | L3 HR | #P |
|---|---|---|---|---|---|
| bt.A | 2 | 0.9 (+/- 0.40) | 0.98 (+/- 0.02) | 0.51 (+/- 0.10) | 6 |
| | 4 | 1.1 (+/- 0.44) | 0.98 (+/- 0.01) | 0.53 (+/- 0.10) | 5 |
| cg.B | 2 | 1.4 (+/- 0.60) | 0.65 (+/- 0.02) | 0.87 (+/- 0.03) | 7 |
| | 4 | 1.5 (+/- 1.05) | 0.66 (+/- 0.02) | 0.87 (+/- 0.02) | 5 |
| ep.B | 2 | 1.0 (+/- 0.01) | 0.99 (+/- 0.00) | 0.98 (+/- 0.04) | 5 |
| | 4 | 1.0 (+/- 0.01) | 0.99 (+/- 0.00) | 1.00 (+/- 0.01) | 5 |
| ft.B | 2 | 0.7 (+/- 1.58) | 0.95 (+/- 0.04) | 0.87 (+/- 0.14) | 6 |
| | 4 | 0.8 (+/- 2.60) | 0.97 (+/- 0.05) | 0.75 (+/- 0.25) | 9 |
| is.B | 2 | 3.0 (+/- 5.77) | 0.83 (+/- 0.13) | 0.75 (+/- 0.33) | 8 |
| | 4 | 4.2 (+/- 10.0) | 0.82 (+/- 0.12) | 0.74 (+/- 0.33) | 4 |
| lu.B | 2 | 1.1 (+/- 0.24) | 0.95 (+/- 0.02) | 0.49 (+/- 0.10) | 4 |
| | 4 | 1.0 (+/- 0.20) | 0.95 (+/- 0.01) | 0.69 (+/- 0.11) | 4 |
| mg.B | 2 | 0.8 (+/- 1.27) | 0.99 (+/- 0.01) | 0.49 (+/- 0.08) | 6 |
| | 4 | 0.9 (+/- 1.90) | 0.99 (+/- 0.01) | 0.46 (+/- 0.08) | 8 |
| sp.A | 2 | 1.6 (+/- 0.25) | 0.96 (+/- 0.00) | 0.47 (+/- 0.05) | 9 |
| | 4 | 2.2 (+/- 0.34) | 0.96 (+/- 0.00) | 0.46 (+/- 0.02) | 7 |
| snp | 2 | 1.0 (+/- 0.09) | 0.95 (+/- 0.02) | 0.06 (+/- 0.07) | 8 |
| | 4 | 0.9 (+/- 0.05) | 0.94 (+/- 0.03) | 0.36 (+/- 0.21) | 6 |
| svm | 2 | 0.9 (+/- 0.32) | 0.91 (+/- 0.05) | 0.48 (+/- 0.21) | 6 |
| | 4 | 1.4 (+/- 0.41) | 0.90 (+/- 0.04) | 0.48 (+/- 0.18) | 5 |
| Avg | | 1.4 (+/- 1.38) | 0.92 (+/- 0.03) | 0.61 (+/- 0.12) | 6 |

2-thread and 4-thread runs. This is computed by first computing the difference between the standard deviation of the baseline and the weighted standard deviation of the phases, and then dividing it by the baseline standard deviation. A large reduction in standard deviation means that the phase analysis succeeds in breaking varying program behavior into homogeneous phases. The results show that when looking at the program's execution in terms of phases that on average the standard deviation for CPI is reduced by 50%, the L2 hit rate by 60% and the L3 hit rate by 45%. The reason why there is little reduction in the standard deviation for `ep.B` for L2 and L3 hit rates is that there was little standard deviation to begin with as shown in Table VI.1.

## VI.D Discovering Phases Across Parallel Runs

One of the motivations for us to perform phase analysis for parallel programs it to be able to examine the same behavior and performance when using a different number of threads. This can be used by programmers and scientists to study where they should tune their code and to better understand the implications of increasing the number of processors to run an application on. For example, to analyze the benefit of parallelizing a program we would like to take a representative slice of the program's execution when using 2 threads, and that same exact slice for 3 threads, 4 threads, etc... and compare how the program's CPI or cache hit rates change as we vary the number of threads. Prior work on the scalability of parallel applications [9, 64, 66] has focused on how the overall execution of the program scales as the number of threads varies. In comparison, we are instead focusing on the scalability of the program in terms of how each of its phases scales as the number of threads varies.

In this section, we describe how we achieve this "thread-varying" phase analysis by extending the parallel phase analysis described in Section VI.C.

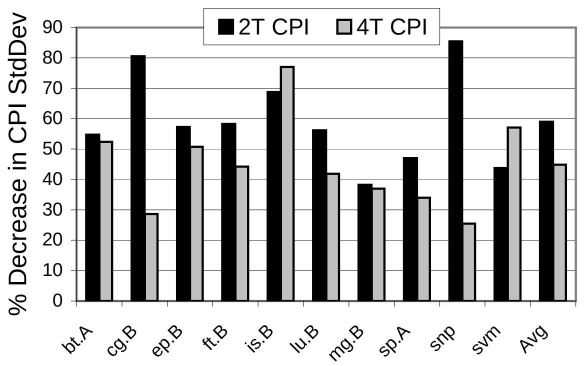


Figure VI.2: Percent reduction in standard deviation for CPI with phase analysis for 2 and 4 threads (T)

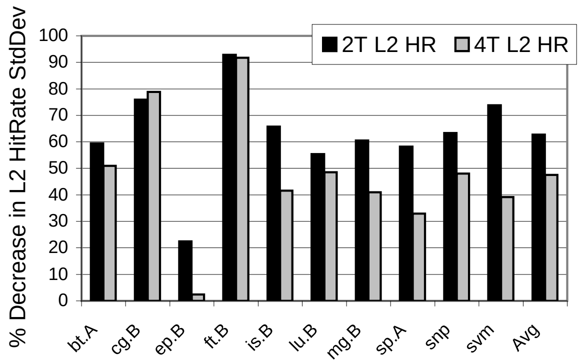


Figure VI.3: Percent reduction in standard deviation for L2 hit rate with phase analysis for 2 and 4 threads (T)

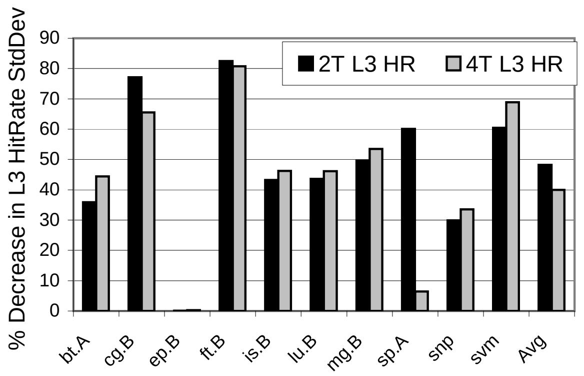


Figure VI.4: Percent reduction in standard deviation for L3 hit rate with phase analysis for 2 and 4 threads (T)

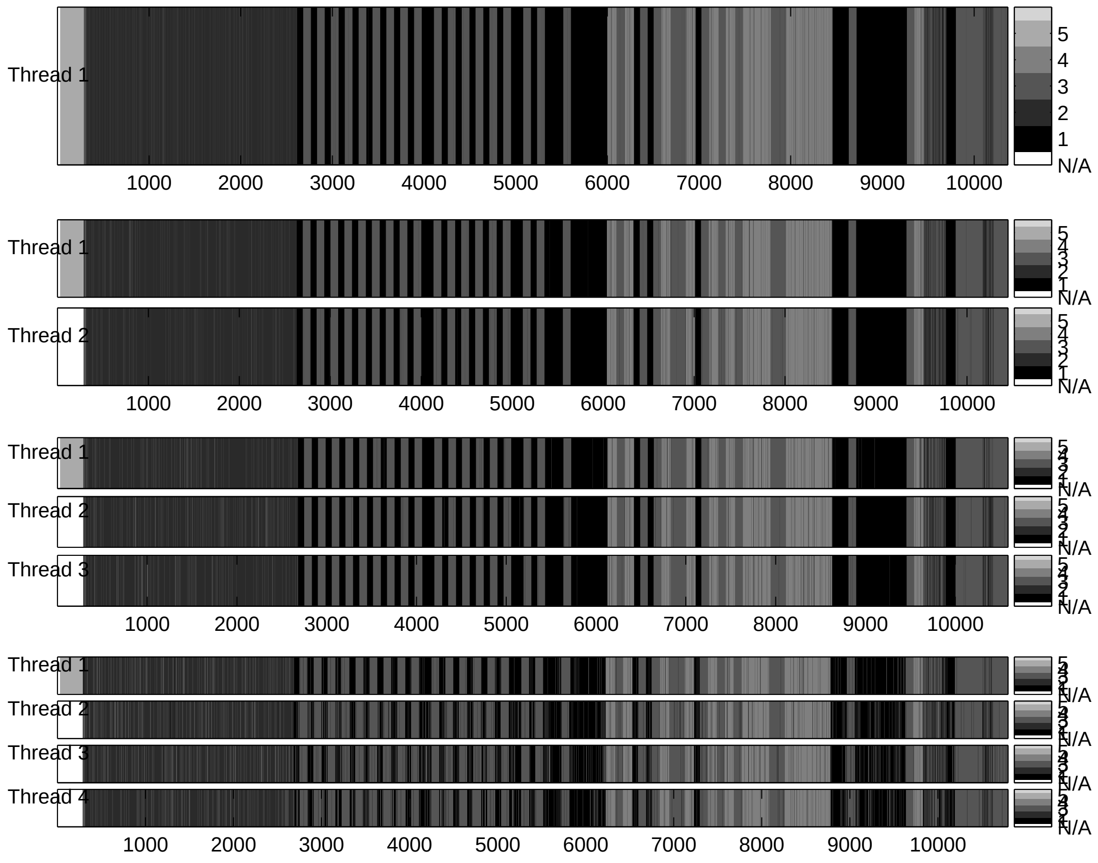


Figure VI.5: Phases discovered across 4 different parallel executions of snp: serial, 2, 3, and 4 threads. The $x$-axis shows the phase classification results over the global count of all instructions executed across all threads.

### VI.D.1 Phase Analysis Varying the Number of Threads

The prior section described how to find phases in a single parallel run of a program. The goal of this analysis is to find similar intervals of execution between different runs as the number of threads is varied. We call this *thread-varying phase analysis*.

We want to perform this analysis on a parallel program running a specific input varying the number of threads. We start by combining all of the sampled BBVs for each thread for a given run as described in Section VI.C. We then concatenate each of these run's sampled BBVs together, where the number of threads has been varied, for a specific parallel binary/input. SimPoint phase analysis is then run over this vector trace. This results in a clustering that successfully groups together not only intervals from separate threads of the same run, as shown in the prior section, but also intervals from different runs, where the number of threads were varied.

### VI.D.2 Thread-Varying Phase Analysis Results

In this work we applied our thread-varying phase analysis on four separate runs for each program: serial, 2 threads, 3 threads, and 4 threads. We combined the runs as described, and computed the phases across the threads and runs. In doing this thread-varying phase analysis, we verified that the intervals grouped within the same phase from the same run had similar architecture metrics as found in the previous section. But intervals from different runs (different number of threads) grouped into the same phase will not have the same architecture metrics. This is exactly what we want to analyze. We use this thread-varying analysis to see for similar code regions how the architecture metrics varied for a phase as the number of threads was varied.

To show this, we will examine the execution of `snp` running with a single thread (serial execution), as well as 2, 3, and 4 threads. We then combine the threads in each of the parallel runs and perform our thread-varying phase analysis. Note that the number of phases chosen is different from Section VI.C since we are performing the thread-varying clustering.

Figure VI.5 shows the phases discovered across four different parallel executions for `snp`. The top most sub-figure displays the phases in a single threaded run of `snp`. The next two sub-figures below show phases in a two threaded run. The next three sub-figures are for a three threaded run, and the last four sub-figures are a four threaded run. The $x$-axis shows the phase classification results over the global count of all instructions executed across all threads. Each phase is denoted by a different color or gray-scale (as shown on the right side of the Figure), and the same phase colors are used across the different runs. White means that no instructions are executed during that interval due to synchronization or serialization.

Figure VI.5 shows that even though each run has a different number of threads, we are able to identify the same regions of execution across the different runs. The initialization phase for `snp` is the shade of the first phase in the single threaded execution. Exactly one thread has that color and all other threads are inactive during that part of execution for the multi-threaded runs. It is also worth noting that the phases also line up along the $x$-axis. In this figure the $x$-axis is the number of instructions retired across all threads in a run, and this means that the phases found between different runs execute a similar number of instructions.

Figure VI.6 shows the number of cycles per phase across 4 different parallel executions of `sp.A`. The $y$-axis is the number of cycles and $x$-axis is the different runs varying the number of threads 1, 2, 3 and 4. In this Figure, the number of cycles ($y$-axis) is the actual time spent executing the benchmark. Cycles are not accounted for on a per-thread basis, as it was in the CPI computation of Section VI.C, but actually represent the time elapsed while one or more threads is in a specific phase. If two or more threads are in two different phases during a part of execution, the time elapsed is split among them with appropriate weights. For example, Phase 2 (the top line) accounts for 11 billion cycles of execution with one thread, and only 3.8 billion cycles when four threads are used.

This Figure is an example of the coherency of our phase definition. Increasing thread-count improves the performance of each phase for this application. An intuitive downward trend is visible for each phase, however the trend varies from phase to phase. One can see that Phase 2 and Phase 4 (the top 2 lines) benefit the most from the parallelization, and one can go back to the code to analyze why this is the case. It also shows that more significant speedups can be achieved by parallelizing Phase 2 and 4 up to four threads, whereas Phase 5 has diminishing returns from parallelization once two threads are used. This is a confirmation that it is worthwhile to perform phase analysis on parallel benchmarks, as each phase exhibits different parallelization potential and performance.

Figure VI.7 shows the number of instructions retired that are classified into each of the phases across 4 different parallel runs (again, 1, 2, 3, and 4 threaded executions). This shows that across the different runs, each phase occupies a similar number of executed instructions. For example, for Phase 2 (the top line), the total number of instructions executed is 80 billion with one thread, and about 85 billion for 4 threads.

The important observation here is that the proportion of intervals assigned to each phase is the same across the different runs when varying the threads. This shows that phase behaviors coherently correspond to the execution of different paths in the code. The thread-count increase changes the distribution of the execution of these paths among different threads, but does not significantly

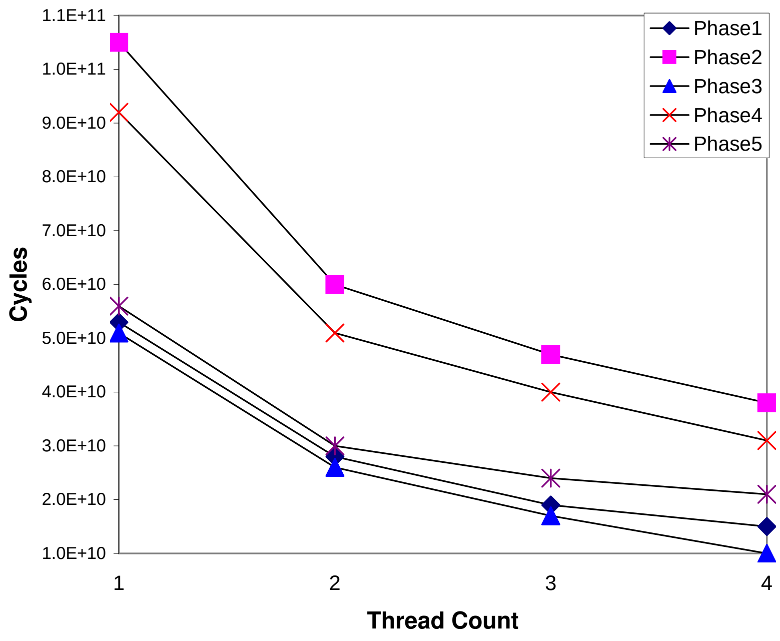


Figure VI.6: Number of cycles per phase across 4 different parallel executions of sp.A: serial, 2, 3, and 4 threads.

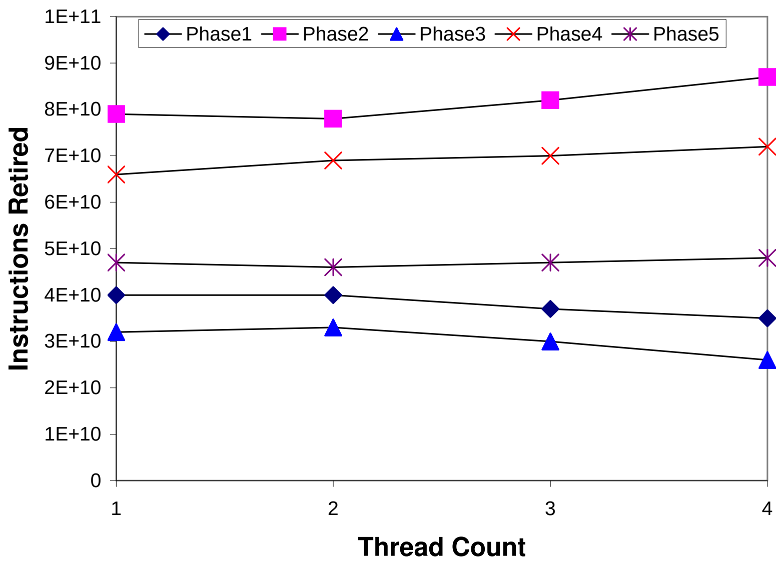


Figure VI.7: Number of instructions per phase across 4 different parallel executions of sp.A: serial, 2, 3, and 4 threads.

alter their nature nor the amount of executed instructions in each phase. It is therefore to be expected and required for a good definition of phases that a given phase behavior occupies approximately the same number of instructions retired, independently from the thread-count, as the number of threads is varied.

## VI.E Parallel SimPoint

We now examine using the phase groupings described in Section VI.C for Parallel SimPoint. The goal is to choose a small set of simulation points (on a per thread basis) that when simulated on a deterministic multi-thread simulator [38, 63] provide an accurate representation of the complete parallel run.

For Parallel SimPoint, we use the phase clustering algorithm described in Section VI.C. A simulation point is chosen for each phase, which is the interval for a specific thread with its sampled basic block vector closest to the centroid of the phase. Remember for our approach, phases can represent intervals from different threads after clustering. Each simulation point is assigned a weight equal to the percent of the program's execution (in terms of intervals) its phase represents (no matter what thread the interval came from). The architecture metrics for these simulation points are then gathered during simulation, and the results are combined with the weights to create an overall estimate of the program's execution in terms of architecture metrics.

Figure VI.8 shows the relative error rates for CPI and L2 hit rate for the 2-thread and 4-thread runs when comparing the parallel SimPoint estimated metric to the overall program's baseline metric. The number of simulation points used for each program is shown in Table VI.1. The results show that the CPI error is 15% or less, with an average of 3% for the 2-thread runs, and similar results are seen for the cache hit rate. Lower error rates are seen for the 4-thread

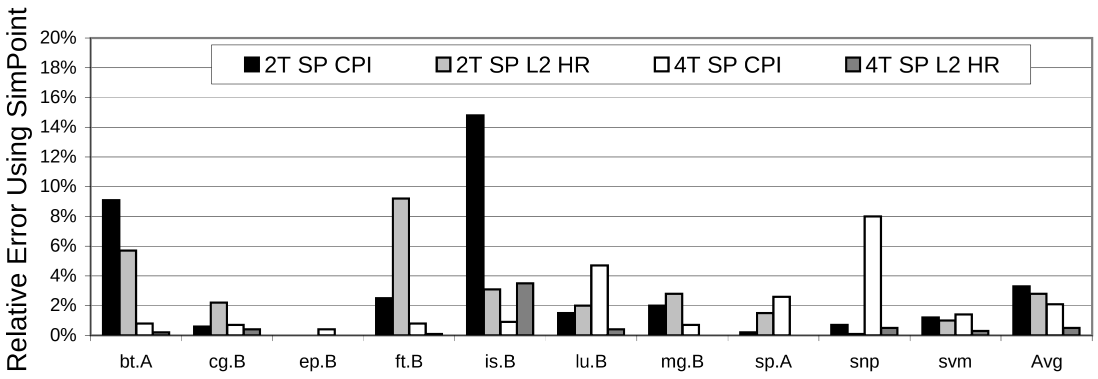


Figure VI.8: SimPoint relative error rates for CPI, L2 and L3 hit rates

runs. Programs like `is.B` have higher error rates due to the huge deviation in program behavior between the parallel and sequential part of execution. Even so, the error rate is small (less than 4%) for 4-threads. This result, along with the reduction in standard deviation as shown in Section VI.C, shows that our approach groups similar parts of execution together based only on the sampled code signatures.

## VI.F Related Work

Efficient simulation techniques are more difficult to apply in parallel execution models. In this section we describe related work that focused on parallel simulation.

Van Biesbrouck et al. [4] used phase behavior to guide simulation for Simultaneous Multithreading [69]. A co-phase matrix is generated to represent the per-thread performance for each potential combination of the single-threaded phase behaviors that can be found when multiple programs are run together. The co-phase matrix is populated by collecting samples of the programs' phase combinations, and is used to guide fast forwarding between samples. This approach is effective in guiding SMT simulation and significantly reduces the simulation time.

Our approach is different in that we are focusing on parallel applications instead of multiple applications running in parallel. Our approach focuses on identifying the unique phase behaviors seen, and then taking one sample (simulation point) of each phase behavior with whatever else is executing with it at that time to represent that phases execution.

## VI.G Summary

In this chapter we focus on discovering phases in parallel applications running on shared memory systems. We start by describing how to recognize similar activities performed by different threads for a program's execution. The results showed that this can be used with SimPoint to accurately represent the program's parallel behavior with an average error less than 4% for CPI, and L2 and L3 hit rates, as well as significantly reduce the standard deviation of these metrics within a phase.

We also showed that we can perform thread-varying phase analysis across different runs of a program as the number of threads used varies from 1 to 4 threads. We found that thread-varying phase analysis can be used to examine the effect on specific parts of the program's execution as the number of threads are varied. This can be used by researchers to better understand a parallel program's execution for different number of threads/processors. Finally we showed that using the parallel phase analysis can be used to accurately pick simulation points to guide multi-threaded simulation.

## Acknowledgements

The text in this chapter is in part a reprint of the material from the paper, *Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder, and Carole Dulong, "Detecting*

*Phases in Parallel Applications on Shared Memory Architectures"*, in the proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006) April 2006, Rhodes Island, Greece. The dissertation author was the primary researcher and author and the co-authors involved in the publication [55] directed, supervised, and assisted in the research which forms the basis for that material.

# VII

# Summary and Future Directions

This dissertation presents techniques for characterizing time varying program behavior for efficient simulation. Understanding the cycle level behavior of a processor running an application is crucial to modern computer architecture research. Cycle detail simulation is prohibitively slow while benchmarks are exceedingly long. By representatively sampling only the unique behaviors we observe in real programs, simulation overhead can be reduced to a reasonable level and achieve very accurate performance estimates.

The main idea behind SimPoint is the realization that programs typically only exhibit a few unique behaviors that are interleaved with one another through time. By finding these behaviors and then determining the relative importance of each one, we can maintain both a high level picture of the program's execution and at the same time quantify the cycle level interaction between the application and the architecture. The key to being able to find these phases in an efficient and robust manner is the development of a metric that can detect the underlying shifts in a program's execution that result in the changes in observed behavior. In this thesis we have discussed one such method of quantifying executed code similarity, and use it to find program phases through the application of unsupervised learning techniques.

SimPoint automates the process of picking simulation points using an off-line phase classification algorithm based on $k$-means clustering, which significantly reduces the amount of simulation time required. Selecting and simulating only a handful of *intelligently* picked sections of the full program provides an accurate picture of the complete execution of a program, which gives a highly accurate estimate of performance. The SimPoint software can be downloaded at:

`http://www.cse.ucsd.edu/users/calder/simpoint/`

For the industry-standard SPEC programs, SimPoint has less than a 6% error rate (2% on average) for the results in this thesis, and is 1,500 times faster on average than performing simulation for the complete program's execution. Because of this time savings and accuracy, our approach is currently used by architecture researchers and industry companies (e.g. [48] at Intel) to guide their architecture design exploration.

In this thesis we presented the following contributions for efficient simulation using the baseline SimPoint model:

- Explore data-mining and statistical advances in doing phase analysis that optimize both the runtime and accuracy of SimPoint as well as target the overall simulation time.
- Present an approach that finds a single set of simulation points to be used across all binaries for a single program. This allows for simulation of the same parts of program execution despite changes in the binary due to ISA changes or compiler optimizations.
- Present a method of characterizing the behavior of parallel applications and use it to pick simulation points to guide multi-threaded simulations.

We now summarize each of these contributions.

### Optimizing Runtime, Accuracy and Simulation Overhead

We explored the primary parameters that have influence on how SimPoint and the k-means clustering algorithm behave. We described how we achieve a reasonable running time for k-means while maintaining a high accuracy by calibrating the number of clustering iterations, number and type of random seeds used, and the number of dimensions needed. We showed that its possible to subsample the vectors during clustering to enable the clustering of very large data sets. We also presented a binary search method over k clusterings that achieves comparable accuracy to an exhaustive search, but at a fraction of the time.

Additionally, we presented the Variance SimPoint algorithm that uses a user defined confidence and probabilistic error bound to guide the picking of $k$. This algorithm first gives priority to choosing a clustering that has well formed (based upon the code frequencies) clusters. This is to make sure that the clustering is representative across different architecture configurations. The Variance SimPoint clustering for picking samples shows that tighter probabilistic error bounds are seen when compared to random sampling when using the same amount of samples as there are simulation points. This is due to Variance SimPoint always choosing a representative sample from each cluster to make sure all the unique/different behavior in the program is captured. In comparison, for a small number of samples, random sampling tends to over/under sample some of the program's behavior resulting in looser probabilistic error bounds.

We also presented the Early SimPoint algorithm whose goal is to find representative simulation points early in a program's execution to reduce fast-forward simulation time. The amount of fast-forward time required for a simulation is the number of instructions it takes to reach the last simulation point for a program/input run. When using an execution interval size of 100 million instructions, we found that Early SimPoint had a 3.6 times shorter fast-forward length on average than the standard SimPoint algorithm. When using an interval size of 10 million, we found the fast-forward length to be only 12% of the full execution on average, with an average CPI error of 4%. These results show that these early simulation points can be used to significantly reduce the time spent fast-forwarding to reach all of the simulation points for a program, while still providing accurate results.

### Variable Length Intervals and Cross Binary Simulation Points

Using real program datasets we showed that the majority of a program's execution is spent in loops. The average number of instructions per loop iteration can change over time. This means that the period lengths are not stable over time. An ideal fixed interval length for one section of execution may be dissonant with another portion of the program's execution. Since no single interval length will do a good job representing the program, we presented the concept of variable length interval lengths that can capture the varying period of the program's behavior pattern. We also modified SimPoint to consume variable length intervals, which is essential for our Cross Binary Simulation Points.

Researchers testing a new ISA extension, examining a new architecture, or trying a new compiler optimization may need to analyze and evaluate performance across different binaries of a program. Due to the slow nature of performance simulators, it has become a standard practice to use representative sampling simulation techniques. We examined two approaches for simulation when there are multiple binaries for a single program/input.

The first approach simply applies the existing SimPoint approach separately on each binary, creating a different set of simulation points for each binary. This approach can accurately estimate the performance for each binary by using different simulation points for each binary, but the approach can have significant error when comparing performance across binaries, since the different simulation points may emphasize different behaviors.

The second approach identifies simulation points that represent the same behaviors across all binary representations of a program. This allows us to simulate the same parts of execution as we change the ISA or compiler optimizations during design space exploration. Our approach finds phase transitions during execution that are identifiable in all of the binaries considered. We use these phase markers along with SimPoint to pick simulation points to represent the full execution of the program, and to identify the exact same start and end of execution for the simulation points in each binary. Our results show that this method does not suffer from changing biases that can occur with the first approach, so cross-binary simulation points can be used to accurately compare performance across binaries.

**Characterizing Parallel Applications**

We presented a method that characterizes phases in parallel applications running on shared memory systems. We showed how to recognize similar activities performed by different threads for a program's execution. The results showed that this can be used with SimPoint to accurately represent the program's parallel behavior with an average error less than 4% for CPI, and L2 and L3 hit rates, as well as significantly reduce the standard deviation of these metrics within a phase.

We also showed that we can perform thread-varying phase analysis across different runs of a program as the number of threads used varies from 1 to 4 threads. We found that thread-varying phase analysis can be used to examine the effect on specific parts of the program's execution as the number of threads are varied. This can be used by researchers to better understand a parallel program's execution for different number of threads/processors. Finally we showed that using the parallel phase analysis can be used to accurately pick simulation points to guide multi-threaded simulation.

## VII.A Future Work

The SPEC CPU 2006 suite executes on the order of a trillion instructions per benchmark. Conservative projections estimate an exponential trend in the SPEC CPU instruction counts, and future generation suites will execute many trillions of instructions per benchmark. At the same time, processors are becoming more intricate and their simulations are running slower. These opposing trends will drive the need for efficient simulation techniques even more in the future.

The granularity of intervals used to split a program execution have a significant impact on the phases discovered in a program. While big intervals capture large scale behavior, smaller intervals can capture detailed regions of complex behavior. In many programs there is some periodic behavior, which the right interval size can cleanly capture.

In SimPoint we want to find this ideal interval size for any given program. This interval size may lead to a cleaner phase characterization and more accurate yet shorter simulation time. Modeling the tradeoff between interval granularity and resulting accuracy and simulation time is needed. Signal processing has many tools that can analyze the program behavior trace and extract useful information regarding fundamental periodic behavior. An analysis to determine an ideal interval size for a particular program would be a significant contribution to SimPoint.

Machine learning is an active field with a continuous flow of improving techniques. SimPoint utilizes several data mining methods, for example clustering with K-Means, dimension reduction with Random Projection, and cluster rating with the Bayesian Information Criterion. New developments in clustering can lead to more accurate simulation samples being picked. One attractive approach is the Expectation-Maximization (EM) soft clustering algorithm which we have started examining. It assigns a probability distribution for a point corresponding to its similarity (distance) for each possible centroid (cluster center) in the data set. This can be very useful if we have an interval spanning multiple behaviors during the execution. The EM algorithm can detect these unique behaviors within the interval and assign the appropriate weight distribution for the interval across multiple phases. There are many other algorithms out there to try that can provide new opportunities for improvement.

The demand for efficient simulation techniques has driven this area of research into the spotlight. With efficient simulation techniques such as SimPoint more researchers are now considering how to exploit these in their methodology. An improved methodology will have far reaching benefits on architecture research. Greater efficiency in evaluating a study will reduce the turnaround time between hypothesis and experimental evaluation. More importantly, the validity of the experimental evaluation will be sound using techniques that accurately represent the complete program-processor execution model. Looking ahead we can expect longer benchmarks and slower simulators as well as new unforeseen challenges. It is crucial for researchers to continue addressing these challenges with efficient simulation techniques.

# Bibliography

[1] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19:716–723, 1974.

[2] M. Annavaram, R. Rakvic, M. Polito, R. Hankins, J.Y. Bouguet, and B. Davies. The fuzzy correlation between code and performance predictability. December 2004.

[3] R. E. Bellman. *Adaptive Control Processes.* Princeton University Press, 1961.

[4] M. V. Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[5] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for uniprocessor and simultaneous multithreading simulation. In *International Conference on High Performance Embedded Architectures and Compilers*, November 2005.

[6] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[7] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[8] G. Casella and R. L. Berger. *Statistical Inference.* Duxbury, Pacific Grove, CA, USA, 2002.

[9] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[10] D. Cheng, A. Gersho, B. Ramamurthi, and Y. Shoham. Fast search algorithms for vector quantization and pattern matching. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 9.11.1–9.11.4, 1984.

[11] Thomas M. Conte. Systematic computer architecture prototyping. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1992.

[12] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 468–477. IEEE Computer Society, 1996.

[13] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, 2000.

[14] S. Dasgupta. How fast is $k$-means? In *COLT*, page 735, 2003.

[15] B. Davies, J.-Y. Bouguet, M. Polito, and M. Annavaram. ipart: An automated phase analysis and recognition tool. Technical report, Microprocessor Research Labs - Intel Corporation, November 2003.

[16] 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. *SIGARCH Comput. Archit. News*, 32(2):350, 2004.

[17] C. Elkan. Using the triangle inequality to accelerate $k$-means. In *ICML*, pages 147–153, 2003.

[18] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. *SIGKDD Explor. Newsl.*, 2(1):51–57, 2000.

[19] U. Fayyad, C. Reina, and P. Bradley. Initialization of iterative refinement clustering algorithms. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 194–198. AAAI Press, 1998.

[20] Richard M. Fujimoto and William B. Campbell. Direct execution models of processor behavior and performance. In *Proceedings of the 19th conference on Winter simulation*, pages 751–758. ACM Press, 1987.

[21] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[22] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine Learning*, 36(1-2):389–422, April 2002.

[23] Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood. Using machine learning to guide architecture simulation. *Journal of Machine Learning Research*, 7:343–378, 2006.

[24] J. Haskins and J. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors (ICCD'01)*, page 32. IEEE Computer Society, 2001.

[25] J. Haskins and J. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation, 2003.

[26] D. Hochbaum and D. Shmoys. A best possible heuristic for the *k*-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.

[27] http://www-mount.ee.umn.edu/ lilja/spec2000/. Minnespec: A new spec benchmark workload for simulation-based computer architecture research.

[28] P. Indyk, A. Amir, A. Efrat, and H H. Samet. Efficient algorithms and regular data structures for dilation, location and proximity problems. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, pages 160–170, 1999.

[29] A. Jaleel, R. S. Cohn, C. Luk, and B. Jacob. Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps. Technical Report UMDSCA-2006-01, Intel, January 2006.

[30] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research, 2002.

[31] AJ KleinOsowski, John Flynn, Nancy Meares, and David J. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. pages 83–100, 2001.

[32] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[33] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[34] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[35] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[36] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[37] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *11th International Symposium on High Performance Computer Architecture*, February 2005.

[38] K. Lepak, H. Cain, and M. Lipasti. Redeeming ipc as a performance metric for multithreaded programs. In *12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[39] W. Liu and M. Huang. EXPERT: Expedited simulation exploiting program behavior repetition. In *International Conference on Supercomputing*, June 2004.

[40] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. June 2005.

[41] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[42] J. McNames. Rotated partial distance search for faster vector quantization encoding. *IEEE Signal Processing Letters*, 7(9), 2000.

[43] A. Moore. The anchors hierarchy: Using the triangle inequality to survive high-dimensional data. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 397–405. AAAI Press, 2000.

[44] http://phase.hpcc.jp/omni/benchmarks/npb/.

[45] Derek B. Noonburg and John Paul Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97)*, page 298. IEEE Computer Society, 1997.

[46] Sebastien Nussbaum and James E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24. IEEE Computer Society, 2001.

[47] Mark Oskin, Frederic T. Chong, and Matthew K. Farrens. HLS: combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA*, pages 71–82, 2000.

[48] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *International Symposium on Microarchitecture*, December 2004.

[49] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. December 2004.

[50] D. Pelleg and A. Moore. $X$-means: Extending $K$-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734, 2000.

[51] M. S. Pepe. The statistical evaluation of medical tests for classification and prediction. *Oxford University Press*, 2003.

[52] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[53] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder. Cross binary simulation points. In *International Conference on Performance Analysis of Systems and Software*, 2007.

[54] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder. ross binary simulation points. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[55] E. Perelman, M. Polito, J. Bouguet, J. Sampson, and B. Calder. Detecting phases in parallel applications on shared memory architectures. In *International Parallel and Distributed Processing Symposium*, 2006.

[56] F. J. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 3(2):131–169, 1999.

[57] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[58] G. Schwarz. Estimating the dimension of a model. *The Annnals of Statistics*, 6(2):461–464, 1978.

[59] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[60] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[61] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[62] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[63] R. Singhal, K.S. Venkatraman, E. Cohn, J.G. Holm, D.Koufaty, M.J. Lin, M. Madhav, M. Mattwandel, N. Nidhi, J. Pearce, and M. Seshadri. Performance analysis and validation of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), February 2004.

[64] Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory parallel systems. *SIGMETRICS Perform. Eval. Rev.*, 22(1):171–180, 1994.

[65] P. Smyth. Clustering using Monte Carlo cross-validation. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, August 1996.

[66] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[67] http://www.intel.com/research/mrl/pnl/.

[68] P.K. Szwed, D. Marques, R.M. Buels, S.A. McKee, and M. Schulz. Simsnap: Fast-forwarding via native execution and application-level checkpointing. In *Proceedings of the 8th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-8)*, Jume 2004.

[69] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture (ISCA)*, June 1995.

[70] D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[71] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *2005 International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*, pages 47–67, November 2005.

[72] http://www.intel.com/software/products/vtune/.

[73] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[74] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture*, June 2003.