UNIVERSITY OF CALIFORNIA SAN DIEGO

Sampled Simulation for Multithreaded Processors

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

in

Computer Science

by

Michael Van Biesbrouck

Committee in charge:

Bradley Calder, Chair
Paul Chau
Lieven Eeckhout
Timothy Sherwood
Michael Taylor
Dean Tullsen

2007

The dissertation of Michael Van Biesbrouck is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California San Diego

2007

DEDICATION

For my family; they made me possible.

EPIGRAPH

Measure with a micrometer.
Mark with a chalk.
Cut with an axe.

Ray's Rule for Precision

Interfere? Of course we should interfere!
Always do what you're best at, that's what I say.

Doctor Who

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation would not have been possible without my advisor, Professor Brad Calder. He guided me in all my research at UCSD and in many ways shaped the material contained herein. I also need to thank my co-authors, Professors Tim Sherwood and Lieven Eeckhout. Tim was involved in the creation of the original co-phase matrix paper; as a final-year PhD student he contributed to the basic ideas that started the paper and as a first-year professor at UCSB he helped write the final published version. When I presented the paper at ISPASS in 2004, I first met Lieven. He began collaborating with me and Brad shortly thereafter. Our first paper idea did not work out, but since then we published four papers together that formed the rest of this dissertation. Lieven contributed substantially to the ideas and writing in each of these papers, meeting weekly by phone despite an awkward time difference.

I must thank all the students in the Architecture Lab, particularly those that kept the machines and submission queues running over the years. During my time at UCSD I have probably simulated a quadrillion instructions using simulators that sacrificed efficiency for accuracy. In the Architecture Lab at UCSD we simulate more instructions than almost anywhere else, just so that others will need to simulate fewer instructions than ever before. This could not be done without the effort of many people that was put into the lab infrastructure, nor without consequences to everyone else trying to run jobs simultaneously, often with pressing deadlines. I think that we would all like to thank the grants that allowed us to get a new cluster of machines when we moved to the new building, making our more extravagant experiments possible.

All of the professors, resident and merely visiting, whose classes, seminars and talks that I attended while at UCSD also contributed. Nothing is ever so off-topic that does not inspire useful ideas. Similarly, this dissertation is the

lesser for every class and talk that I did not attend, however good or bad the reason. Fortunately, others around me at UCSD did attend and benefited directly, and through them I benefited indirectly.

As a foreign student who moved, newly married, to the part of the mainland US furthest from where he lived and into unfamiliar geography (desert, ocean, mountains and the scrubland in between), I would like to thank the family members, co-workers and friends that supported my move and the nice people at my destination, on campus and off. I also need to thank Irwin and Joan Jacobs for making it financially possible.

Chapter II and Chapter IV contain material from *A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation* [59], in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Michael Van Biesbrouck, Timothy Sherwood and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are ©2004 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Chapter III contains material from *Efficient Sampling Startup for Sampled Processor Simulation* [55], in *IEEE Micro Magazine*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are ©2005 Springer-Verlag Berlin Heidelberg. Free use of this material is permitted under the German Copyright Law of September 9, 1965, in its current version (amended 8 May 1998). Non-free uses may require permission from Springer-Verlag.

Chapter III contains material from *Efficient Sampling Startup for SimPoint* [57], in *IEEE Micro Magazine*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are ©2006 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Chapter IV and Chapter V contain material from *Considering All Starting Points for a Simultaneous Multithreading Simulation Methodology* [56], in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are ©2006 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Chapter V contains material from *Representative Multiprogram Workloads for Multithreaded Processor Simulation* [58], in *IEEE International Symposium on Workload Characterization (IISWC)*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are ©2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

VITA

1997	Bachelor of Mathematics in Computer Science University of Waterloo
1999	Masters of Mathematics in Computer Science University of Waterloo
2007	Doctor of Philosophy in Computer Science University of California, San Diego

ABSTRACT OF THE DISSERTATION

Sampled Simulation for Multithreaded Processors

by

Michael Van Biesbrouck
Doctor of Philosophy in Computer Science
University of California San Diego, 2007
Professor Bradley Calder, Chair

Microarchitectural simulation of multithreaded architectures with shared resources, such as simultaneous multithreading (SMT) cores and multi-core processors with shared caches, is time-consuming and the results of simulation may be difficult to interpret. It is time-consuming because modern benchmarks run for hundreds of billions (or even trillions) of instructions, and accurate multi-core and SMT simulation requires higher-detail models than single-threaded simulation. The statistics collected when two programs execute together can be difficult to interpret because the programs both exhibit independent phase behavior and affect each other's execution. Starting one program slightly later than during the original execution will change the phases that execute together and thus change the effects that the programs have on each other.

Accurate sampled simulation requires accurate sample collection. We evaluate techniques to improve sampling accuracy and performance, both for single-threaded and multithreaded simulation. These techniques include warming the CPU with detailed execution, storing cache state and techniques to minimize the size of checkpoints.

Previous work showed that single-program performance can be accurately estimated by dividing execution into phases and only simulating represen-

tative samples from each phase. We demonstrate that the juxtaposition of phases ('co-phase') from a pair of programs has similar behavior to a single-threaded phase. Furthermore, simulation of all possible co-phases allows analysis of all distinct SMT behaviors and this comprehensive knowledge of program interactions can be combined with information about the sequence of phases executed by each program to reconstruct the combined execution of the programs from any given starting point. Given the short samples, the set of executions from all possible starting offsets can be sampled in minutes, determining the average performance of the programs. This removes the problem of interpreting the results of small numbers of experiments.

Finally, we propose three techniques for using the co-phase techniques to summarize the behavior of all possible interactions within a suite of benchmarks. We reduce the scale of this problem using Priciple Components Analysis, allowing our techniques to scale to large numbers of benchmarks an concentrate simulation on the most significant behaviors.

I

Introduction

Treatments of single-threaded simulation methodology are legion, but prior to the work included in this dissertation there were few papers on multithreaded simulation methodology. Chapter III details improvements to single-threaded simulation methodology, but all subsequent chapters address the problems of multithreaded simulation.

This dissertation shows that naïve experimentation using multithreaded simulation is slow and may produce misleading results. To remedy these problems we present the *co-phase matrix* method of sampled multithread simulation, improvements to sampled simulation in general, and techniques to avoid misleading results by looking at all possible ways that benchmarks can interact on a multithreaded processor.

This introduction outlines the state of the multithreaded processor market and the problems inherent in efficient, accurate and meaningful multithreaded simulation. Each aspect of the problem is treated in full in subsquent chapters.

I.A Multithreaded Commodity Processors

The number of transistors available in commodity CPU chip design continues to increase, roughly doubling every 18 months [33, 45]. In the past, the increased transistor count was used to double performance through increasingly aggressive out-of-order execution designs. This created more on-chip execution resources than could be used simultaneously by a single execution thread. *Simultaneous Multithreading (SMT)* [54] was developed to take advantage of these wasted resources. A single SMT core provides the same interface as a shared-memory machine with multiple cores; each virtual core is called a *context*. Several programs (or threads of a single program) execute simultaneously, sharing most aspects of the underlying microarchitecture while keeping independent architected state. Since an SMT machine can usually fully utilize microarchitectural resources it is worthwhile to make the chip design even more aggressive. This increases the potential for single-threaded performance as well as aggregate multithreaded performance. The ALPHA EV8 design included support for four contexts but was never built. Later, Intel extended the Pentium4 processor to support two contexts and called this implementation of SMT *Hyper-Threading*[32].

Despite optimism that single-CPU performance would continue to increase [23], the high complexity of CPU designs and resulting power and heat problems made it difficult to maintain the rate of single-threaded performance improvements. Intel abandoned the Pentium4 in favor of the descendents of its earlier architecture, still in use on laptops, and thus stopped creating Hyper-Threaded CPUs. This capability is slated to return to Intel processors in 2008, but in a different and unspecified form [50]. The focus of most chip designers switched to chip multicore processor (CMP) design. Using multiple, less powerful, cores on a die is a simpler and more power-efficient way to use increased transistor counts to improve aggregate performance. Unlike SMT processors, CMPs

normally only share caches and off-chip communication bandwidth (particularly to memory).

IBM introduced the POWER5, using both SMT and CMP technology. Each chip contained two out-of-order cores with two contexts each. Intended for mainframes, multiple chips were put in packages, multiple packages on cards and potentially many cards in a single machine. The POWER6, successor to the POWER5, continues to use SMT and CMP technology. To achieve higher performance and lower power use, the cores are now in-order. The simpler core designs allow smaller, faster and more efficient cores. Previously, under contract to design the processors for Xbox 360 and PlayStation3[1] game consoles, IBM created similar in-order SMT cores with two contexts. The Xbox 360 has three such cores; the PlayStation3 only has one but it shares a die with many other cores of a different type.

Sun Microsystems also has multicore processors with a variant of SMT. The Niagara processor has eight small, in-order cores with four contexts each. Its successor, the Niagara2, supports eight contexts per core. This line of processors targets webserver workloads with complicated flow control and frequent memory dependence. For these processors, the contexts exist just to ensure that the processors rarely stall by providing several applications between which to switch. The reported performance of the Niagara2 is impressive, but the intended applications of the processor and the differences in SMT implementation put it outside the scope of this work.

This dissertation focuses on two-context out-of-order SMT processors. Dual-core single-context processors and in-order cores are simpler cases of the SMT processors examined so all SMT results can be applied directly. Where appropriate, extensions to processor configurations with more than two concurrent

[1]The PlayStation3's CPU is also known as the CELL processor and can be found in a variety of other configurations, such as supercomputers and network processors.

threads are discussed.

I.B Single-Threaded Sampled Simulation

Benchmarks used for single-threaded simulation are frequently too long to simulate in their entirety. Current industry-standard benchmarks, such as SPEC CPU2000 execute hundreds of billions of instructions; SPEC CPU2006 is designed to have significantly longer runtimes, and has programs that run for trillions of instructions. Even on today's fastest architectural simulators, simulating some of the SPEC CPU2000 benchmarks takes several weeks to complete. The processors to be simulated are becoming slower to simulate as they become more complex and the length of the benchmarks to be simulated is increasing so simulation time is increasing even though the computers running the simulations are becoming faster. In order to measure cycle-level events and to examine the effect that microarchitecture optimizations would have on the whole program, computer architects usually avoid long detailed simulation times by taking samples, either a small number of large execution intervals or a large number of small ones, and then use that information to approximate the full program behavior.

Early papers used single execution intervals near the start of program execution. For programs that change behavior over time, this could be extremely inaccurate.[48] Statistical sampling methods such as random sampling and periodic sampling [65] avoid this problem by sampling many intervals throughout all of execution. SimPoint [19, 48, 39] uses targeted sampling to identify small numbers of samples by analyzing program execution and dividing it into several distinct *phase behaviors* and simulating just one interval from each phase. In general, the technique is more accurate with many phases and small intervals than it is with fewer phases and larger intervals. SimPoint is discussed in more detail in Section II.A.

These techniques can be used to accurately estimate the performance characteristics of each benchmark on a given microarchitectural configuration. They are most accurate in comparing two program executions when the same set of sampled intervals are used in each case. Uniprocessor simulators are normally designed so that every execution of a program produces the same results and executes all of the same non-speculative instructions no matter how the simulated microarchitecture is configured. Simulators sample execution using checkpoints or fast-forwarding to minimize the time required run benchmarks. Techniques to maximize the accuracy of checkpoints while reducing disk-usage requirements are discussed in Chapter III.

I.C Multithreaded Sampled Simulation

When multiple threads are simulated then determinism is typically lost. Even on a uniprocessor, full-system simulators and simulators supporting thread libraries may simulate instructions in different orders and the number of simulated instructions may change. This dissertation focuses on multithreaded simulators without full-system support and workloads of single-threaded progams. The non-speculative instructions executed are deterministic, as in the uniprocessor case, but the instructions being run concurrently will change. Changing the simulated microarchitecture alters the performance of the simulated workload, but the programs will be affected to different degrees. Consider a multiprocessor as a uniprocessor that fetches bundles of instructions drawn from all the workload programs simultaneously (instead of a bundles of instructions from a single program). It is a deterministic uniprocessor executing a nondeterministic instruction stream. Runs with different microarchitectural parameters effectively run different programs, thus making them hard to compare.

Not only are multithreaded workloads more complicated, the simulators

must be more complicated to execute them with accurate timing. Uniprocessor simulators always know when instruction arguments will be available as soon as an instruction is fetched, allowing immediate execution of the instruction effects without loss of simulation fidelity. In contrast, the timing of an instruction in a multithreaded machine may depend upon instructions from a different program that have yet to be fetched. For example, a load may need to wait hundreds of cycles to execute because its address is dependent upon a previous load that did not hit in cache. The additional time that it will take the dependent load to complete after the load address is available depends upon which cache (if any) contains the data. If the data is currently in a shared cache then any of tens to hundreds of upcoming instructions in other programs could evict the cache line, nearly doubling the already long exection time of the dependent load. On an SMT processor, even slight variations in timing will affect contention for functional units and other resources. Faithful reproduction of these effects requires accurate simulation of complex processor internals that could be safely ignored in uniprocessor simulation, increasing simulation time.

Questions that were simple to answer in a single-threaded environment, such as "which sections of execution will represent the complete workload?", are more complex in a multithreaded environment. In a multithreaded processor we care about the sections of execution coming from two separate programs and which sections will execute at the same time. For single-threaded execution, the choice of intervals to execute is dependent upon a simple analysis of the benchmarks alone, but in the multithreaded case the combinations of intervals that execute at the same time is dependent also on the microarchitectural configuration.

When two or more programs share a processor's resources at a cycle-level granularity, as is the case with SMT, the performance of the two applications

becomes entangled; this is true to a lesser extent for programs that only share a cache. If there are multiple programs running at the same time, the behavior of all the programs will affect not only the overall performance of the machine but also the distribution of performance between the different programs, causing some to execute faster than others. Changing a hardware parameter that has an effect on performance may change which parts of the programs execute together. This change, in turn, may mean that the machine is now executing a different mix of behaviors, which will influence the overall performance. This interdependence, or entanglement, makes it difficult to summarize or estimate the overall behavior of the system. The challenge in creating a sampling approach to SMT lies in determining how far to fast-forward each individual thread between samples. This distance will vary as the threads execute through different phases of execution; the distance also varies with different microarchitecture configurations.

This dissertation solves the problem using the *co-phase matrix*. In Chapter IV the method is explained and used to track the execution of pairs of programs from given starting points for 1–10 billion instructions with high levels of accuracy.

I.D Multithreaded Workloads

Most of today's processors run multiple programs simultaneously through SMT or multicore processing. This dissertation does not consider large-scale parallel scientific applications. That type of workload is sufficiently complex that entire supercomputers have been designed for single applications or particular types of applications. In such cases algorithms and architecture may be co-designed for maximal performance of a single program running by itself. For this class of applications Perelman *et al.* [40] have a phase-based analsysis procedure for programs running on real hardware and Ekman and Stenström [15] use ran-

dom sampling for programs that synchronize frequently. Instead, we consider the performance of commodity machines that must be capable of running a wide variety of applications and the mixed workloads that run on machines that are not dedicated to a single task. Some programs are designed to run identical threads simultaneously for CPU-intensive tasks such as encoding full-motion video, but most users will run a heterogeneous set of programs and individual programs may run many threads that accomplish distinct tasks. For example, a web browser and its helper applications may be simultaneously parsing HTML, decoding JPEG images, decompressing gzip-encoded web pages, interpreting JavaScript, playing audio and showing Flash animations while other programs and operating system threads run on the same system. A thorough analysis of a new system requires understanding the consequences of running any combination of these threads and programs.

Virtualization provides additional types of mixed workloads. Now, multiprocessors are frequently bought to run several operating systems simultaneously. The scale can range from a home user wanting to run Windows and Linux at the same time to datacenter server consolidation in which one large machine runs many servers for several companies. In any such case we expect the workloads running in the separate operating system images to have little or no coordination between themselves. Within a datacenter hosting servers for multiple companies there is no reason to believe that the workloads would be similar. Architects designing machines for such uses need to be aware of all possible interactions between virtualized images and their relative importance.

All of the multithreaded work in this dissertation uses the assumption that the workload will consist of independent programs executing with shared resources. For performance-modeling, we need to look at all the ways in which programs in a workload might interact. Programs are not like competitive runners

that both start from the beginning of their track and will run until they reach the end of the track at nearly the same time. They are more like cars and trucks on a highway, entering and exiting based on their private agendas, interacting as they switch lanes and cause traffic jams due to careless driving. To understand how a pair of programs can interact we need to know what happens when they start to run at the same time, when one is half-way through executing as the other starts, and every other combination of points of progress. Furthermore, the set of programs that are concurrently scheduled changes over time so an aggregate view of the performance of an entire benchmark suite is important. These issues are considered in Chapter V.

II

Background

There are two types of background material presented in this chapter. First, we explain the SimPoint tools for the analysis of single-threaded programs. These tools are used in several different ways in each of the subsequent chapters. Second, we make a brief overview of techniques for evaluating multithreaded processors that are related to this work.

II.A SimPoint

SimPoint is a toolset for the analysis of single-threaded dynamic program execution. It determines which parts of execution are similar to each other and finds representative samples for each program behavior so that detailed execution of the representative samples is sufficient to accurately estimate program execution.

To perform the SimPoint analysis, a desired sample size is chosen. The sample size is the interval size (in dynamic instructions) at which the user wants to perform simulation or program analysis. We use sample sizes of 1, 5 and 10 million instructions for different experiments in this dissertation; some papers have used samples as large as 300 million instructions. The program execution is

then broken up into consecutive intervals equal to the sample size, and a profile is gathered for every interval. The profile measures what code was executed during each interval and its frequency. Either an instrumented binary running on real hardware or a simulator doing fast functional simulation can collect this data. The profile is represented as *Basic Block Vectors (BBVs)*. A basic block vector is an array with one entry for every static basic block in the program. An interval's basic block vector has a count for each basic block, which is the number of times that the basic block was executed during the interval. A given program will have different BBV profiles for different inputs, but for a given program binary and input the profile will be independent of the microarchitecture used.

The angle between two BBVs determines how different the relative mix of basic blocks is between two intervals, as does the distance between them. The large number of basic blocks in the program, and often even those within a single interval, makes the comparison time-consuming. Creating a random projection from the original dimensional space to a smaller dimensional space decreases the cost of comparing two vectors. If two vectors are similar before projection, they will still be similar after the projection. Using a sufficiently large dimensional space makes it unlikely for dissimilar vectors from the old space to be projected onto similar vectors in the new space.

In the Single SimPoint process, the BBVs for every interval are compared to a BBV representing the complete execution of the program in order to find a single interval that is the closest to the complete execution of the program. This technique is used by the simulation methodologies described in Section II.B.4 and Section II.B.6.

More generally, SimPoint can be used to classify each interval as having one of several distinct program behaviors, called *phase behaviors*. Each set of intervals with the same phase behavior make up a *phase*. SimPoint determines

which intervals to group into phases using the k-means clustering algorithm, which divides the BBVs into k groups. The groups are chosen to be centered around k points. Many such clusterings are possible, so the algorithm minimizes the sum of squares of distances between BBVs and cluster centers. Choosing k too high will increase the number of phases without improving the model, so SimPoint tries many values of k and picks one that accurately models the distribution of BBVs. The *Bayesian Information Criterion (BIC)* is used to compare the quality of clusterings with different numbers of cluster centers. This metric combines the quality of the clustering with a cost function for increasing k so that small accuracy improvements are not bought at the cost of large increases in the number of phases.

Finally, SimPoint finds intervals that best represent each phase. It does this by choosing the interval with BBV closest to the centroid of all the BBVs in the phase. Simulating this interval, called a *simulation point* will give an execution representative of the phase. Collecting statistics for all of the simulation points and weighting them according to the number of intervals in their phases allows us to estimate whole-program performance. Since the phases are microarchitecture-independent we can safely compare microarchitectures using this approach. In general, the most accurate results can be had by using more phases, requiring more intervals to be simulated. Shorter intervals also lead to more phases. Fortunately, many phases represented by short intervals can be faster to simulate than fewer large intervals while still having improved accuracy [39, 19].

II.B Multithreaded Performance Evaluation Approaches

Evaluating the performance of a multithreaded machine running a single multithreaded benchmark is simple in concept — run a program from start to

finish and time how long it takes. When multiple programs are run together they will end at different times but the system behavior running a single program might not be of interest. Stopping the timing after the first program exits will not represent system performance, either, since two systems could have the first program end at the same time but make different amounts of progress in the other programs. In a simulation environment, the programs will execute so slowly that complete execution is impossible. Solving this problem is the main contribution of this thesis. In this section we survey the approaches that have been used elsewhere.

II.B.1 SPEC Rate Metrics

The SPEC rate metrics evaluate system performance when running multiple copies of the same benchmark on multithreaded machines. Since all of the running programs are the same and the objective is to run them as many times as possible within a lengthy time limit, different ending times are not an issue. We can use the techniques in Chapter IV to efficiently evaluate this metric.

II.B.2 Weighted Speedup

The *weighted speedup* metric [51] can be used when programs make different amounts of progress on a multithreaded machine. Each program's performance improvement (speedup) is measured by dividing its multithreaded performance by its single-threaded performance for the same sequence of instructions. The speedups are summed to get the weighted speedup. This provides the exact aggregate speedup over the instructions that were executed as compared to single-threaded execution. This number is useful for studies that simulate program execution on a multithreaded processor for a fixed number of instructions and on real hardware for long-running workloads that are not identically mixed

each time.

Although a good tool, weighted speedup can be misleading. A processor or scheduler can maximize its weighted speedup by executing programs unfairly, giving more resources to the programs that will get the greatest gains. Two weighted speedups from different executions could represent effective workloads that are too dissimilar to be worth comparing. We propose methods to evaluate performance that are not dependent upon execution termination to avoid these problems.

II.B.3 Variation in Multithreaded Performance

Alameldeen and Wood [1] have examined program variability causing nondeterminism. For example, memory latency can be variable but is modeled as a constant in most simulators; they have observed that small variations in memory latency can lead to widely different execution paths in multithreaded systems. They observed this phenomenon using both real and simulated CMP machines. Their experiments show that variability due to nondeterminism plays a significant role unless simulations are continued for long periods of time. They use a statistical approach to reduce the simulation times of their real-world benchmarks. By repeating short experiments many times they are able put a tight bound on the average-case execution times and avoid situations in which unpredictable events could cause them to reach incorrect conclusions.

In Chapter V we find that small changes is program offsets can lead to different multithreaded performance results. Rather than sampling many slight variations in starting offsets we look at the effect of starting offset on a much larger scale; the variations due to small changes are subsumed in the variation on larger scales. We also use moderately long sample lengths to minimize the possible effects of variation.

II.B.4 Using One SimPoint Per Benchmark

Raasch and Reinhardt [42] examined the effects of partitioned resources on SMT execution. To reduce simulation time they used 100M-instruction single simulation points from SimPoint for each benchmark program-input pair. As a further optimization, *Principle Components Analysis (PCA)* and clustering were used to eliminate simulation points that were similar to other simulation points. The executed all of the possible two-thread pairs until both programs executed at least 100M instructions or one thread executed 300M instruction; most executions would have gone beyond the range of the simulation point interval. Their goal was to capture the typical behavior of a workload independent of when each individual program was started, ensuring that they observed a wide range of program behaviors using a limited amount of simulation time.

In Chapter V we use similar techniques to represent the behaviors of benchmark suites but we use multiple phases per benchmark and weight the results according to phase frequency over the benchmark suite. Our work in Chapter IV allows us to efficiently examine execution from all possible starting points without losing any phase behaviors.

II.B.5 Repeated Full Execution

Tuck and Tullsen [53] ran multithreaded workloads on real SMT hardware to compare practice with simulation papers. Not being constrained by slowdown from simulations they ran each multi-program combination multiple times in succession. Timing began when a thread terminated for the first time and did not end until at least 12 threads were run to completion, more if necessary to ensure that three instances of each thread were included in the timing. This methodology allows averaging over multiple offsettings and avoids timing execution with only one thread running. Repeated runs of this procedure had an

average maximum variance of 1%.

This technique can be recreated efficiently using our techniques in Chapter IV, but we prefer to randomly sample the effects of different starting offsets. This allows us to statistically analyze when we have looked at enough of the possible program executions to bound our error.

II.B.6 FAME

The *FAME* methodology [60, 61] consists of different proposals for execution on real hardware and in simulators, published subsequently to most of the results presented in this dissertation. On real hardware they advocate the previous technique of repeatedly executing the programs together. For simulation purposes, they use 300M-instruction single SimPoints, similar to those used by Raasch and Reinhardt [42]. As with the real-hardware methodology, they execute the samples repeatedly together. In both cases, they attempt to iterate until the average performance converges. They predict the required number of repetitions required to ensure this will happen based on single-threaded performance, assuming that single-threaded behavior will dominate over thread interactions.

Our differences from the previous two techniques necessarily apply here. Additionally, our predictions of convergence are based on statistical tests rather than heuristics.

II.C Acknowledgement

This chapter contains material from *A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation* [59], in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Michael Van Biesbrouck, Timothy Sherwood and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are

III

Sampling Startup

Before looking at multithreaded sampled simulation, we need to look at the techniques that can make single-threaded sampled simulation accurate and efficient. Some of them will be reused later as part of the multithreaded simulation methodology.

III.A Introduction

Modern computer architecture research relies heavily on cycle-accurate simulation to help evaluate new architectural features. In order to measure cycle-level events and to examine the effect that hardware optimizations would have on the whole program, architects are forced to execute only a small subset of the program at cycle-level detail and then use that information to approximate the full program behavior.

The subset chosen for detailed study has a profound impact on the accuracy of this approximation, and picking these points so that they are as *representative* as possible of the full program is a topic of several research studies [8, 27, 47, 48, 65]. The two difficulties in using these sampling techniques efficiently and accurately are getting the correct memory image to execute the

sample and warm architecture state when simulating the sample. We collectively refer to both issues as *Sampling Startup*.

III.A.1 Sample Starting Image

The first issue to deal with is how to accurately provide a sample's starting image. The *Sample Starting Image* (SSI) is the state needed to accurately emulate and simulate the sample's execution to achieve the correct output for that sample[1]. The two traditional approaches for providing the SSI are fast-forwarding and using checkpoints. Fast-forwarding quickly emulates the program's execution from the start of execution or from the last sample to the current sample. The advantage of this approach is that this is trivial for all simulators to implement. The disadvantage is that it serializes the simulation of all of the samples for a program, and it is non-trivial to have a low-overhead fast-forwarding implementation—most fast-forwarding implementations in current simulators are fairly slow.

Checkpointing is the process of storing the program's image right before the sample of execution is to start. This is similar to storing a core dump of the program so that it can be replayed at that point in execution. A checkpoint stores the register contents and the memory state prior to a sample. The advantage of checkpointing is that it allows for efficient parallel simulation. The disadvantage is that if a full checkpoint is taken it can be huge and consume too much disk space and take too long to load.

In this chapter we examine two efficient ways of storing the SSI. One is a reduced checkpoint where we only store in the checkpoint the words of memory that are to be accessed in the sample we are going to simulate. This is the *Touched Memory Image (TMI)*. Our TMI files are two orders of magnitude smaller than

[1] For convenience of exposition, we use 'sample' as a noun to refer to a sampling unit and 'sample' as a verb to refer to collecting a sample unit.

normal checkpoints. The second approach is very similar, but is represented differently. For this approach we store a sequence of executed load values for the complete sample, the *Load Value Sequence (LVS)*; after compression it becomes the *Reduced Load Value Sequence (RLVS)*. Both of these approaches take about the same disk space. Since they are small they also load instantaneously and are significantly faster than using fast-forwarding and full checkpoints.

III.A.2 Sample Architecture Warmup

Once we have an efficient approach for dealing with the sample's starting image we also need to reduce as much error in simulation due to the architecture components not being in the same state as if we simulated the full detailed execution from the start of the program up to that simulation point. To address this we examine a variety of previously proposed techniques and compare them to storing the detailed state of the memory hierarchy as a form of architecture checkpoint.

We first examine a technique called *Hit on Cold* which assumes that all architecture components are cold and the first access to it during the sample's simulation is a hit. A second technique we study uses a *Fixed Warmup* period before the execution of each sample. Recently, more sophisticated warmup techniques [11, 20, 21, 22] have focused on finding for each sample how far back in the instruction stream to go to start warming up the architecture structures. We examine the performance of MRRL [21, 22] in this chapter. An important advantage of this type of technique is its accuracy. The disadvantage is that it requires architecture component simulation for N million instructions before detailed simulation of the sample, which adds additional overhead to simulation.

The final technique we examine is storing an architecture checkpoint of the major architecture components at the start of the sample. This *Architecture*

Checkpoint is used to faithfully recreate the state of the major architecture components, such as caches, TLBs and branch predictors at the start of the sample. It is important that this approach works across different architecture designs for it to be used for architecture design space explorations. To that end, we examine a form of architecture checkpointing that allows us to create the smaller size instances of that architecture component. For example, you would create an architecture checkpoint of the largest cache you would look at in your design space exploration study, and the way we store the architecture checkpoint will allow smaller sizes and associativities to be faithfully recreated.

III.B Background

This section discusses prior work on Sample Startup techniques. We discuss checkpointing and fast-forwarding for obtaining a correct SSI, and warmup techniques for obtaining an architecture checkpoint as accurately as possible.

III.B.1 Sample Starting Image

As stated in the introduction, starting the simulation of a sample is much faster under checkpointing than under fast-forwarding (especially when the sample is located deep in the program's execution trace—fast-forwarding in such a case can take several days). The major disadvantage of checkpoints however is their size; they need to be saved on disk and loaded at simulation time. The checkpoint reduction techniques presented in this chapter make checkpointing a much better alternative to fast-forwarding as will be shown in the evaluation section of this chapter.

Szwed *et al.* [52] propose to fast-forward between samples through native hardware execution, called direct execution, and to use checkpointing to communicate the application state to the simulator. The simulator then runs

the detailed processor simulation of the sample using this checkpoint. When the end of the sample is reached, native hardware execution comes into play again to fast-forward to the next simulation point, *etc.* Many ways to incorporate direct hardware execution into simulators for speeding up the simulation and emulation systems have been proposed, see for example [9, 17, 34, 46].

One requirement for fast-forwarding through direct execution is that the simulation needs to be run on a machine with the same ISA as the program that is to be simulated. One possibility to overcome this limitation for cross-platform simulation would be to employ techniques from dynamic binary translation methods such as just-in-time (JIT) compilation and caching of translated code, as is done in Embra [63], or through compiled instruction-set simulation [37, 43]. Adding a dynamic binary compiler to a simulator is a viable solution, but doing this is quite an endeavor, which is why most contemporary out-of-order simulators do not include such functionality. In addition, introducing JITing into a simulator also makes the simulator less portable to host machines with different ISAs. The code to save and restore checkpoints, however, can be easily portable.

Related to this is the approach presented by Ringenberg *et al.* [44]. They present intrinsic checkpointing, which takes the SSI image from the previous simulation interval and uses binary modification to bring the image up to state for the current simulation interval. Bringing the image up to state for the current simulation interval is done by comparing the current SSI against the previous SSI, and by providing fix-up checkpointing code for the loads in the simulation interval that see different values in the current SSI versus the previous SSI. The fix-up code for the current SSI then executes stores to put the correct data values in memory.

Our approach is easier to implement as it does not require binary modification. In addition, combining intrinsic checkpointing with warmup is not

straight-forward. The fix-up code which stores values in memory affects the cache contents. As a result, implementing intrinsic checkpointing for small simulation intervals can be inaccurate.

III.B.2 Warmup

There has been a lot of work done on warmup techniques, or approximating the hardware state at the beginning of a sample. This work can be divided roughly in three categories: (*i*) simulating additional instructions prior to the sample, (*ii*) estimating the cache miss rate in the sample, and (*iii*) storing the cache content or taking an architecture checkpoint. In the evaluation section of this chapter, we evaluate four warmup techniques. These four warmup techniques were chosen in such a way that all three warmup categories are covered in our analysis.

Warmup N Instructions Before Sample

The first set of warmup approaches simulates additional instructions prior to the sample to warmup large hardware structures [8, 65, 20, 21, 22, 12, 7, 25, 31, 36]. A simple warmup technique is to provide a fixed-length warmup prior to each sample. This means that prior to each sample, caches and branch predictors are warmed by, for example, 1 million of instructions. MRRL [21, 22] on the other hand, analyses memory references and branches to determine where to start warming up caches and branch predictors prior to the current sample. MRRL examines both the instructions between the previous sample and the current sample and the instructions in the sample to determine the correct warmup period. BLRL [12], which is an improvement upon MRRL, examines only the sample to see how far one needs to go back before the sample for accurate warmup.

SMARTS [65] uses continuous warmup of the caches and branch predictors between two samples, *i.e.*, the caches and branch predictor are kept warm by simulating the caches and branch predictor continuously between two samples. This is called functional warming in the SMARTS work. The reason for supporting continuous warming is their small sample sizes of 1000 instructions. Note that continuously warming the cache and branch predictor slows down fast-forwarding.

The warmup approaches from this category that are evaluated in this chapter are fixed-length warmup and MRRL.

Estimating the Cache Miss Rate

The second set of techniques does not warm the hardware state prior to the sample but estimates which references in the sample are cold misses due to an incorrect sample warmup [28, 64]. These misses are then excluded from the miss rate statistics when simulating the sample. Note that this technique in fact does no warmup, but rather estimates what the cache miss rate would be for a perfectly warmed hardware state. Although these techniques are useful for estimating cache miss rate under sampled simulation, extending these techniques to processor simulation is not straight-forward because we need to know which of the excluded misses would have been hits on a warm cache. The hit-on-cold approach evaluated in this chapter is another example of cache miss rate estimation; the benefit of hit-on-cold over the other estimation techniques is its applicability to detailed processor simulation.

Checkpointing the Cache Contents

Lauterbach [30] proposes storing the cache tag contents at the beginning of each sample. This is done by storing tags for a range of caches as they are

obtained from stack simulation. This approach is similar to the Memory Hierarchy State (MHS) approach presented in this chapter (see Section III.C.3 for more details on MHS). However, there is one significant difference. We compute the cache content for one single large cache and derive the cache content for smaller cache sizes. This is more efficient than running a stack simulation as is done by Lauterbach.

Although this can be done through stack simulation, it is still significantly slower and more disk space consuming than simulating only one single cache configuration as we do. The Memory Timestamp Record (MTR) presented by Barr *et al.* [3] is also similar to the MHS proposed here. The MTR allows for the reconstruction of the cache and directory state for multiprocessor simulation by storing data about every cache block. The MTR is largely independent of cache size, organization and coherence protocol. Unlike MHS, its size is proportional to program memory. This prior work did not provide a detailed comparison between their architectural checkpointing approach and other warmup strategies; in this chapter, we present a detailed comparison between different warmup strategies.

III.B.3 SMARTS and TurboSMARTS

Wunderlich *et al.* [65] provide SMARTS, an accurate simulation infrastructure using statistical sampling. SMARTS continuously updates caches and branch predictors while fast-forwarding between samples of size 1000 instructions. In addition, it also warms up the processor core before taking the sample through the detailed cycle-by-cycle simulation of 2000 to 4000 instructions.

At the same time we completed the research for our chapter, TurboSMARTS [62] presented similar techniques that replace functional warming with a checkpointed SSI and checkpointed architectural state similar to what we discuss in this chapter. In addition to what was studied in [62], we compare a

number of reduced checkpointed SSI techniques, we study the impact of wrong-path load instructions for our techniques, and we examine the applicability of checkpointed sampling startup techniques over different sample sizes.

III.B.4 Parallel simulation of simulation points

The most obvious way to get simulation speedup when simulating samples is to employ parallel simulation on a cluster of machines [36, 30, 18, 10]. For example, Girbal *et al.* [18] present DiST to increase the simulation speed through parallel simulation. DiST divides the complete program execution in a number of chunks that are distributed over a number of machines. DiST then fast-forwards on each machine to the assigned chunks. When the assigned chunk is reached, detailed processor simulation gets started. To deal with the warmup issues, DiST simulates the first part of a given chunk twice: once as the first part of the given chunk under cold start, and once at the end of the previous chunk assigned to another machine. Warmup is stopped when both simulations of the same chunk parts converge to identical simulation results.

The results from this chapter are directly applicable to parallel simulation to further improve its speed. The reduced checkpoints can be used to start the simulation on each machine. Note that the reduced checkpoints not only eliminate fast-forwarding time, but are also fast to transfer over a distributed environment due to their small size. The Memory Hierarchy State (MHS) warmup strategy can be used to accurately and efficiently warmup hardware state.

III.C Discussion

Detailed cycle-by-cycle simulation of complete benchmarks is practically impossible due to the huge dynamic instruction counts of today's benchmarks (often several hundred billions of instructions), especially when multiple processor

configurations need to be simulated during design space explorations. Sampling is an efficient way for reducing the total simulation time. There exist two main ways of sampling, statistical sampling and phase-based sampling. The use of SimPoint for phase-based sampling was previously discussed in Section II.A. In this chapter we focus on studying the applicability of the Sample Startup techniques presented for SimPoint and statistical sampling. In addition, we also provide summary results for applying these Sample Startup techniques to SMARTS.

III.C.1 Statistical Sampling

Statistical sampling takes a number of execution samples across the whole execution of the program, which are referred to as clusters in [8] because they are groupings of contiguous instructions. These clusters are spread out throughout the execution of the program in an attempt to provide a representative cross-cut of the application being simulated. Conte *et al.* [8] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC, branch and data cache statistics).

SMARTS [65] provides a version of SimpleScalar [5] using statistical simulation, which uses statistics to tell users how many samples need to be taken in order to reach a certain level of confidence. One consequence of statistical sampling is that tiny samples are gathered over the complete benchmark execution. This means that in the end the complete benchmark needs to be functionally simulated, and for SMARTS, the caches and branch predictors are warmed through the complete benchmark execution. This ultimately impacts the overall simulation time.

III.C.2 Sample Starting Image

The first issue to deal with to enable efficient sampled simulation is to load a memory image that will be used to execute the sample. The *Sample Starting Image* (SSI) is the program memory state needed to enable the correct functional simulation of the given sample.

Full Checkpoint

There is one major disadvantage to checkpointing compared to fast-forwarding and direct execution for providing the correct SSI. This is the large checkpoint files that need to be stored on disk. Using many samples could be prohibitively costly in terms of disk space. In addition, the large checkpoint file size also affects total simulation time due to loading the checkpoint file from disk when starting the simulation of a sample and transferring over a network during parallel simulation.

EIO Files and Checkpointing System Calls

Before presenting our two approaches to reduce the checkpoint file size, we first detail our general framework in which the reduced checkpoint methods are integrated. We assume that the program binary and its input are available through an EIO file during simulation. We use compressed SimpleScalar EIO files; this does not affect the generality of the results presented in this chapter, however. An EIO file contains a checkpoint of the initial program state after the program has been loaded into memory. Most of the data in this initial program image will never be modified during execution. The rest of the EIO file contains information about every system call, including all input and output parameters and memory updates associated with the calls. This keeps the system calls exactly the same during different simulation runs of the same benchmarks.

In summary, for all of our results, the instructions of the simulated program are loaded from the program image in the EIO file, and the program is not stored in our checkpoints. Our reduced checkpoints focus only on the data stream.

Touched Memory Image

Our first reduced checkpoint approach is the *Touched Memory Image (TMI)* which only stores the blocks of memory that are to be accessed in the sample that is to be simulated. The TMI is a collection of chunks of memory (touched during the sample) with their corresponding memory addresses. The TMI contains only the chunks of memory that are read during the sample. Note that a TMI is stored on disk for each sample. At simulation time, prior to simulating the given sample, the TMI is loaded from disk and the chunks of memory in the TMI are then written to their corresponding memory addresses. This guarantees a correct SSI when starting the simulation of the sample. A small file size is achieved by using a sparse image representation, so regions of memory that consist of consecutive zeros are not stored in the TMI. In addition, large regions of non-zero sections of memory with only a few zeros between them are combined and stored as once chunk. This saves storage space in terms of memory addresses in the TMI, since only one memory address needs to be stored for a large consecutive data region.

An optimization to the TMI approach, called the *Reduced Touched Memory Image* (*RTMI*), only contains chunks of memory for addresses that are read before they are written. There is no need to store a chunk of memory in the reduced checkpoint in case that chunk of memory is written prior to being read. A TMI, on the other hand, contains chunks of memory for all reads in the sample.

Load Value Sequence

Our second approach, called the *Load Value Sequence (LVS)*, involves creating a log of load values that are loaded into memory during the execution of the sample. Collecting an LVS can be done with a functional simulator or binary instrumentation tool, which simply collects all data values loaded from memory during program execution (excluding those from instruction memory and speculative memory accesses). When simulating the sample, the load log sequence is read concurrently with the simulation to provide correct data values for non-speculative loads. The result of each load is written to memory so that, potentially, speculative loads accessing that memory location will find the correct value. The LVS is stored in a compressed format to minimize required disk space. Unlike TMI, LVS does not need to store the addresses of load values. However, programs often contain many loads from the same memory addresses and loads with value 0, both of which increase the size of LVS without affecting TMI.

In order to further reduce the size of the LVS, we also propose the *Reduced Load Value Sequence (RLVS)*. For each load from data memory the RLVS contains one bit, indicating whether or not the data needs to be read from the RLVS. If necessary, the bit is followed by the data value, and the data value is written to the simulator's memory image at the load address so that it can be found by subsequent loads; otherwise, the value is read from the memory image and not included in the RLVS. Thus the RLVS does not contain load values when a load is preceded by a load or store for the same address or when the value would be zero (the initial value for memory in the simulator). This yields a significant additional reduction in checkpoint file sizes. An alternate structure that accomplishes the same task is the first load log presented in [35].

III.C.3 Warmup

We compare five warmup strategies, not performing any warmup, hit on cold, 1M-instructions of detailed execution fixed warmup, MRRL and stored architecture state. The descriptions in this section summarize the warmup techniques in terms of how they are used for uniprocessor architecture simulation.

No Warmup

The no-warmup strategy assumes an empty cache at the beginning of each sample, *i.e.* assumes no warmup. Obviously, this will result in an overestimation of the number of cache misses, and by consequence an underestimation of overall performance. However, the bias can be small for large sample sizes. This strategy is very simple to implement and incurs no runtime overhead.

Hit on Cold

The *hit on cold* strategy also assumes an empty cache at the beginning of each sample but assumes that the first use of each cache block in the sample is always a hit. The no warmup strategy, on the other hand, assumes a miss for the first use of a cache block in the sample. Hit on cold works well for programs that have a high hit rate, but it requires modifying the simulator to check a bit on every cache miss. If the bit indicates that the cache block has yet to be used the sample then the address tag is added to the cache but the access is considered to be a hit.

An extension to this technique is to try to determine the overall program's average hit rate or the approximate hit rate for each sample, then use this probability to label the first access to a cache block as a miss or a hit. We did not evaluate this approach for this dissertation.

Memory Reference Reuse Latency

The *Memory Reference Reuse Latency (MRRL)* [21, 22] approach proposed by Haskins and Skadron builds on the notion of memory reference reuse latency. The memory reference reuse latency is defined as the number of dynamic instructions between two consecutive memory references to the same memory location. To compute the warmup starting point for a given sample, MRRL first computes the reuse latency distribution over all the instructions from the end of the previous sample until the end of the current sample. This distribution gives an indication about the temporal locality behavior.

MRRL subsequently determines w_N which corresponds to the $N\%$ percentile over the reuse latency distribution, the point at which $N\%$ of memory references will be made to addresses last accessed within w_N instructions. Warming then gets started w_N instructions prior to the beginning of the sample. The larger $N\%$, the larger w_N, and thus the larger the warmup. We use $N\% = 99.9\%$ as proposed in [21].

Sampled simulation under MRRL then proceeds as follows. The first step is to fast-forward to or load the checkpoint at the starting point of the warmup simulation phase. From that point on until the starting point of the sample, functional simulation is performed in conjunction with cache and branch predictor warmup, *i.e.* all memory references warm the caches and all branch addresses warm the branch predictors. When the sample is reached, detailed processor simulation is started for obtaining performance results. The cost of the MRRL approach is the w_N instructions that need to be simulated under warmup.

Memory Hierarchy State

The fourth warmup strategy is the *Memory Hierarchy State (MHS)* approach, which stores cache state so that caches do not need to be warmed

at the start of simulation. The MHS is collected through cache simulation, *i.e.* functional simulation of the memory hierarchy. Design-space exploration may require many different cache configurations to be simulated. Note that the MHS needs to be collected only once for each block size and replacement policy, but can be reused extensively during design space exploration with smaller-sized memory hierarchies. Our technique is similar to trace-based construction of caches, except that storing information in a cache-like structure decreases both storage space and time to create the cache required for simulation. In addition to storing cache tags, we store status information for each cache line so that dirty cache lines are correctly marked.

Depending on the cache hierarchies to be simulated, constructing hierarchies of *target caches* for simulation from a single *source cache* created by functional simulation can be complicated, so we explain the techniques used and the necessary properties of the source cache.

If a target cache i has s_i sets with w_i ways, then every cache line in it would be contained in a source cache with $s' = c_i s_i$ sets and $w' \geq w_i$ ways, where c_i is a positive integer. We now describe how to initialize the content of each set in the target cache from the source cache. To initialize the contents of the first set in the target cache we need to look at the cache blocks in the first c_i sets of the source cache, giving us $c_i w'$ ways. For a cache with LRU replacement we need to store a sequence number for the most recent use of each cache block. We select the most recently used w_i cache blocks to put in the target set. The next c_i sets of the source cache initialize the second set of the target cache, and so forth. In general, $s' = \texttt{LCM}_i(s_i)$ and $w' = \texttt{max}_i(w_i)$ ensure that the large cache contains enough information to initialize all of the simulated cache configurations. (In the common case where s_i is always a power of two, the least common multiple (LCM) is just the largest such value.)

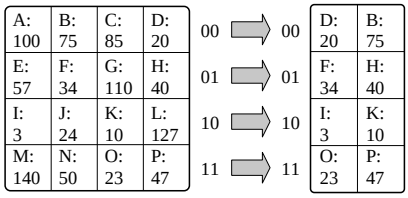

Figure III.1: Reducing associativity from 4-way to 2-way.

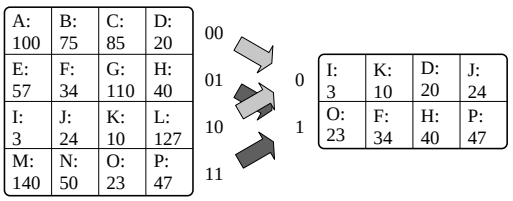

Figure III.2: Reducing the number of sets.

Figures III.1 and III.2 demonstrate how MHS works when reducing the cache associativity and the number of sets, respectively. In the figures, each row is a cache set with a number of columns equal to the associativity of the cache. Each cache block is labeled with a letter; the letters represent tags. The number below the letter is the time (in memory operations) since the cache block was last used. In the top row of the large cache (cache set 00), blocks D and B are the most recently used blocks. When associativity is reduced to two in Figure III.1, those are the two most recently used blocks, so they are retained. Figure III.2 reduces the number of cache lines to two, merging cache sets 00 and 10 to a single set, 0; cache sets 01 and 11 are merged into set 1. The new cache sets contain the most recently used entries from both of the cache sets that were merged into them. Also note that this operation will increase the length of the cache tags by one bit.

Inclusive cache hierarchies can be initialized easily as just described, but exclusive cache hierarchies need something more. For example, assume that the L2 cache is 4-way associative and has the same number of cache sets as a 2-way associative L1 cache. Then the 6 most recently accessed blocks mapped to a single cache set will be stored in the cache hierarchy, 2 in the L1 cache and 4 in the L2 cache. Thus the source cache must be 6-way associative. If, instead, the L2 cache has twice as many sets as the L1 cache then the cache hierarchy will contain 10 blocks that are mapped to the same L1 cache set, but at most 6 of

these will be mapped to either of the L2 cache sets associated with the L1 cache set. The source cache still only needs to be 6-way associative.

We handle exclusive cache hierarchies by creating the smallest (presumably L1) target cache first and locking the blocks in the smaller cache out of the larger (L2, L3, *etc.*) caches. Then the sets in the larger cache can be initialized. Also, the associativity of the source cache used to create our MHS must be at least the sum of the associativities of the caches within a target cache hierarchy.

Unified target caches can be handled by collecting source cache data as if the target caches were not unified. For example, if there are target IL1, DL1 and UL2 caches then data can be collected using the same source caches as if there were IL2 and DL2 caches with the same configuration parameters as the UL2 cache. Merging the contents of two caches into a unified target cache is straight-forward. The source caches must have a number of cache sets equal to the LCM of all the possible numbers of cache sets (both instruction and data) and an associativity at least as large as that of any of the caches. Alternately, if all of the target cache hierarchy configurations are unified in the same way then a single source cache with the properties just described can collect all of the necessary data.

Comparing MHS versus MRRL, we can say that they are equally micro-architecture-independent. Then MHS traces store all addresses needed to create the largest and most associative cache size of interest. Similarly, MRRL goes back in execution history far enough to also capture the working set for the largest cache of interest. The techniques have different tradeoffs, however: MHS requires additional memory space compared to MRRL, and MRRL just needs to store where to start the warming whereas MHS stores a source cache. In terms of simulation speed, MHS substantially outperforms MRRL as MHS does not need to simulate instructions to warm the cache as done in MRRL—loading the MHS

trace is done very quickly. As simulated cache sizes increase, MHS disk space requirements increase and MRRL warming times increase.

The techniques discussed in this section can also be extended to warmup for TLBs and branch predictors. For 1M-instruction simulation points, we only consider sample architecture warmup for caches. We found that the branch predictor did not have a significant effect until we used very small 1000-instruction intervals with SMARTS. When simulating the tiny SMARTS simulation intervals we checkpointed the state of branch predictor prior to each sample. While we can modify TLB and cache checkpoints to work with smaller TLBs and caches, we cannot yet do this for general branch predictors. For experiments requiring branch predictor checkpoints for varying predictor configurations it is necessary to simulate several branch predictors concurrently while collecting checkpoints.

Large, complex conditional branch predictors may need to be stored to disk for each of their possible configurations. If the components of a branch predictor can be considered separately, it may be possible to look at many configurations with only a few variations saved to disk. Branch target buffers and return address stacks can be resized, just like caches, so only one large instance of each needs to be stored. Other branch predictor components may need to be simulated for each size and algorithm used, but they can be combined with target buffers and address stacks of any size.

Situations in which separate microarchitectural components affect each other are more complex to handle. For example, consider an experiment in which cache sizes are varied and several different prefetchers with internal state are examined. The prefetchers affect the contents of the cache, so an ideal simulation would simulate the two structures together, storing correlated cache and prefetcher contents to disk. In the worst case, every combination of two components that interfere with each other would need to be stored to disk. In this case,

Table III.1: Processor simulation model.

I Cache	8k 2-way set-assoc., 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-assoc., 32 byte blocks, 2 cycle latency
L2 Cache	1Meg 4-way set-assoc., 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Pred	hybrid 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	up to 8 inst. per cycle, 128 entry re-order buffer
Func Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV

our technique to resize the caches is not affected by the existence of prefetched data from a single prefetcher. Thus, we can store one copy of the MHS for each prefetcher design even though many cache configurations will be examined. Any microarchitectural structure that does not interfere with prefetching and cache contents can have its configurations simulated separately, minimizing the number of microarchitectural combinations that need to be simulated to create the MHS.

We now evaluate Sample Startup for sampled simulation. After discussing our methodology, we then present a detailed error analysis of the warmup and reduced checkpointing techniques. We subsequently evaluate the applicability of the reduced checkpointing and warmup techniques for both phase-based sampling as done in SimPoint and statistical sampling.

III.C.4 Methodology

We use the MRRL-modified SimpleScalar simulator [21], which supports taking multiple samples interleaved with fast-forwarding and functional warming. Minor modifications were made to support (reduced) checkpoints. We simulated SPEC 2000 benchmarks compiled for the Alpha ISA and we used reference inputs for all of these. The binaries we used in this study and how they were compiled

can be found at `http://www.simplescalar.com/`. The processor model assumed in our experiments is summarized in Table III.1.

For these results we used SimPoint with an interval size of 1 million and Max K set to 400. Figure III.3 shows the number of 1M-instruction simulation points per benchmark. This is also the number of checkpoints per benchmark since there is one checkpoint needed per simulation point. The number of checkpoints per benchmark varies from 15 (`crafty`) up to 369 (`art`). In this chapter we focus on small, 1M-instruction intervals because SimPoint and statistical sampling are most accurate when many (at least 50 to 100) intervals are used. This is most practical when small (1M instructions or less) intervals are accurately simulated. However, we found that the reduction in disk space is an important savings even for 10M and 100M interval sizes when using a reduced load value trace.

For statistical sampling, we consider 50 1M-instruction sampling units randomly chosen from the entire benchmark execution. We also experimented with a larger number of sampling units, however, the results were quite similar.

III.C.5 Error Analysis

Our error analysis covers both phase-based and statistical sampling using all of the warmup techniques previously discussed; we follow this with a brief discussion of the effects of wrong-path loads on LVS.

Phase-based Sampling Using SimPoint

We now study the accuracy of the reduced warmup and checkpointing techniques for phase-based sampling using SimPoint. Figure III.4 shows the accuracy of SimPoint while assuming perfect sampling startup. The average error is 1.3%; the maximum error is 4.8% (`parser`).

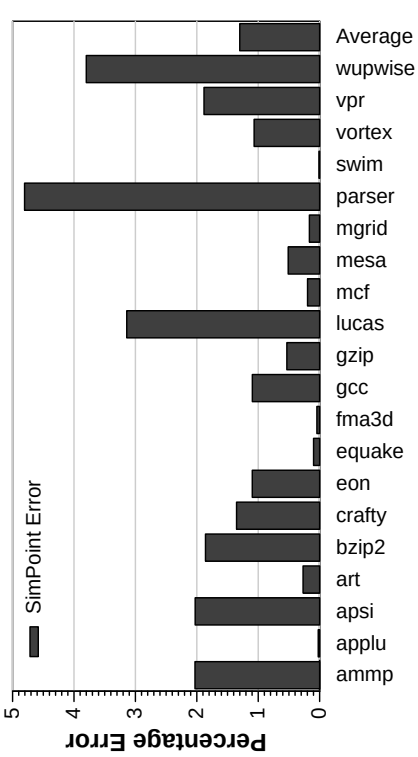

Figure III.4: Accuracy of SimPoint assuming perfect sampling.

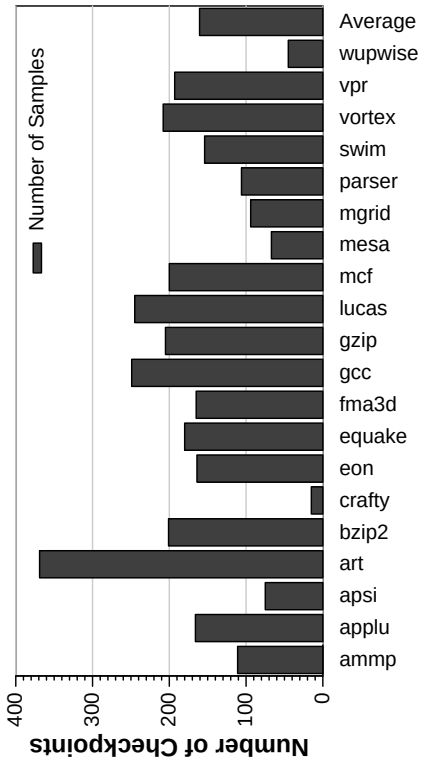

Figure III.3: Number of simulation point samples used with Max K set to 400.

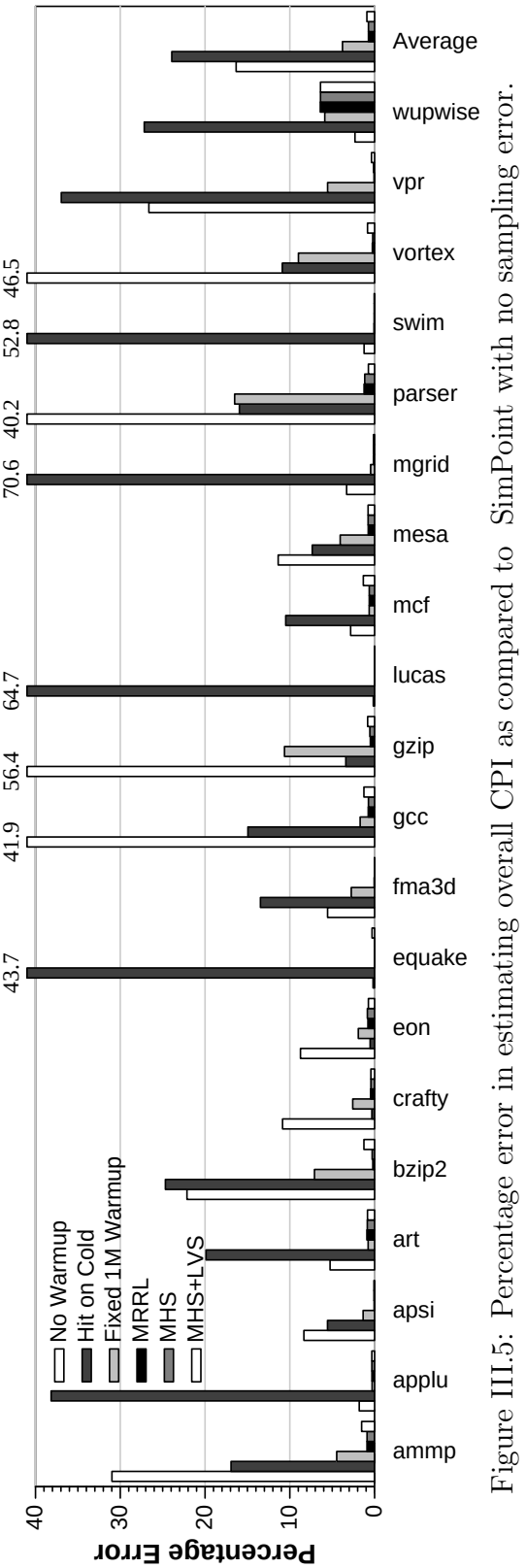

Figure III.5: Percentage error in estimating overall CPI as compared to SimPoint with no sampling error.

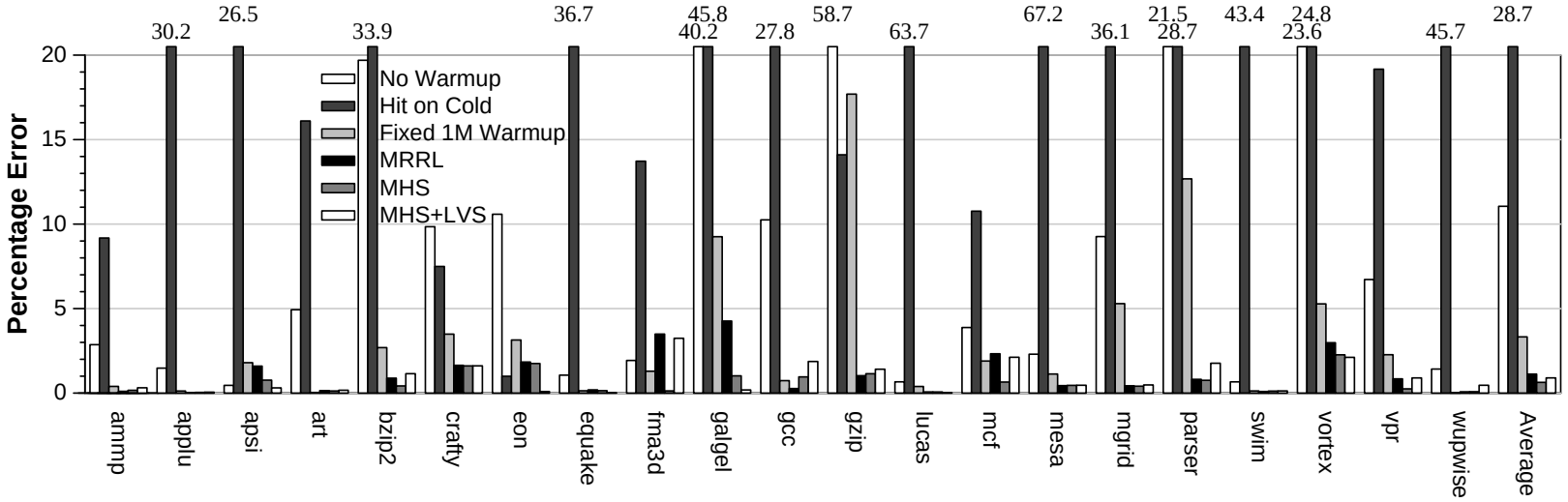

Figure III.6: Average CPI error: average CPI sample error as a percentage of CPI.

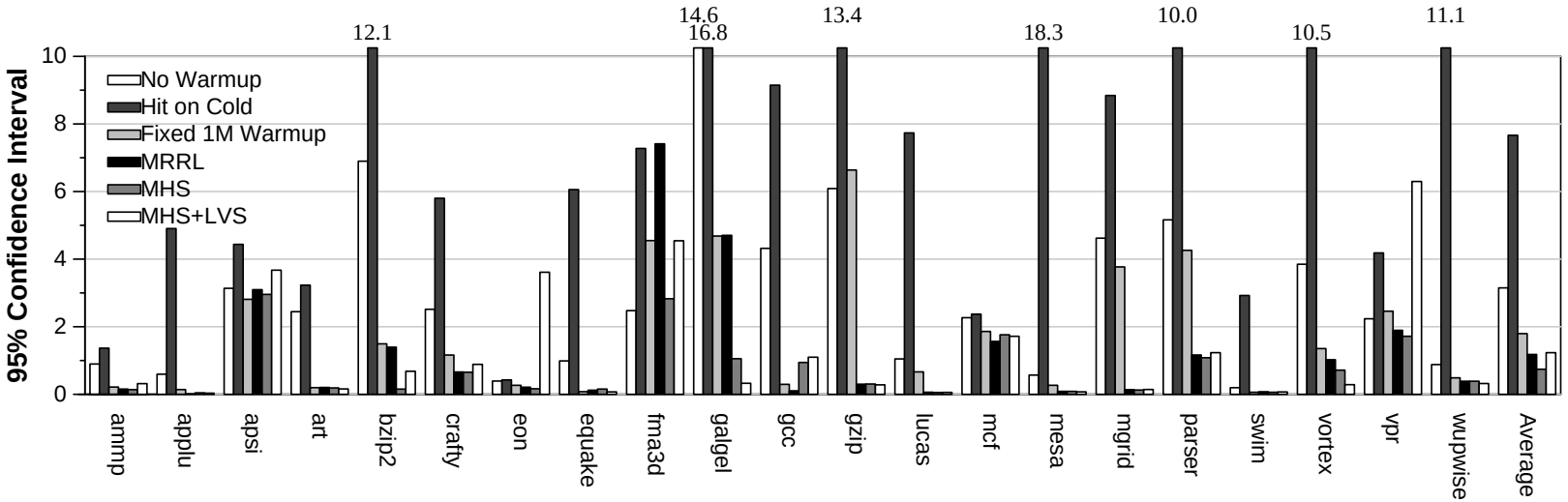

Figure III.7: The 95% confidence interval as a percentage of CPI.

Figure III.5 evaluates the CPI error rates for various Sample Startup techniques as compared to the SimPoint method's estimate using perfect warmup — this excludes any error introduced by SimPoint. This graph compares four sample warmup approaches: no warmup, hit-on-cold, fixed 1M warmup, MRRL and MHS. The no warmup and hit-on-cold strategies result in high error rates, 16% and 24% on average. For many benchmarks one is dramatically better than the other, suggesting that an algorithm that intelligently chooses one or the other might do significantly better. The fixed 1M warmup achieves better accuracy with an average error of 4%; however the maximum error can be fairly large, see for example for `parser` (17%). The error rates obtained from MRRL and MHS are significantly better. The average error for both approaches is 1%. As such, we conclude that in terms of accuracy, MRRL and MHS are equally accurate when used in conjunction with SimPoint.

The error results discussed so far assumed full checkpoints. Considering a reduced checkpoint technique, namely LVS, in conjunction with MHS increases the error rates only slightly, from 1% to 1.2%. This is due to the fact that LVS does not include load values for loads being executed along mispredicted paths.

Statistical Sampling

We first provide a detailed error analysis of our warmup techniques as well as the reduced checkpointing techniques under statistical sampling.

The *average CPI error* is the average over all samples of the relative difference between the CPI through sampled simulation with full warmup, versus the CPI through sampled simulation with the warmup and reduced checkpoint techniques proposed in this chapter. Our second metric is the *95% confidence interval* for average CPI error. Techniques that are both accurate and precise will have low average CPI error and small confidence interval widths.

Figures III.6 and III.7 show the average CPI error and the 95% confidence interval, respectively. In both cases they are expressed as a percentage of the correct CPI. The various bars in these graphs show full SSI checkpointing along with a number of architectural warmup strategies (no warmup, hit on cold, fixed 1M warmup, MRRL and MHS), as well as a reduced SSI checkpointing technique, namely LVS, in conjunction with MHS. We only present data for the LVS reduced SSI for readability reasons; we obtained similar results for the other reduced SSI techniques. In terms of error due to warmup, we find that the no-warmup and hit-on-cold and strategies perform poorly while fixed 1M-instruction warmup usually (but not always) performs reasonably well. MRRL and MHS on the other hand, are shown to perform equally well. The average error is less than a few percent across the benchmarks. Although the per-benchmark results are not always the same as the results using SimPoint, in most cases they are comparable and the overall results are nearly identical.

In terms of error due to the starting image, we see the error added due to the reduced SSI checkpointing is very small. Comparing the MHS bar (MHS with full checkpointing) versus the MHS+LVS bar, we observe that the error added is very small, typically less than 1%. These results are similar to those found with SimPoint.

Wrong-path Loads

The reason for the additional error when using LVS is that under reduced SSI checkpointing, load instructions along mispredicted paths might potentially fetch wrong data from memory since the reduced checkpointing techniques only consider on-path memory references. In order to quantify this we refer to Figures III.8 and III.9. Figure III.8 shows the percentage of wrong-path load instructions being issued relative to the total number of issued loads; this figure

also shows the percentage of issued wrong-path loads that fetched incorrect data (compared to a fully checkpointed simulation) relative to the total number of issued loads. This graph shows that the fraction of wrong-path loads that are fetching uncheckpointed data is very small, 2.05% on average. Figure III.9 then quantifies the difference in percentage CPI error due to these wrong-path loads fetching uncheckpointed data. We compare the CPI under full checkpoint versus the CPI under reduced checkpoints. The difference between the error rates is very small, under 1% of the CPI.

III.C.6 Total simulation time

Figure III.10 shows the total simulation time (in minutes) for the various Sample Startup techniques when simulating all simulation points on a single machine. This includes fast-forwarding, loading (reduced) checkpoints, loading the Memory Hierarchy State and warming structures by functional warming or detailed execution, if appropriate.

The SSI techniques considered here are fast-forwarding, checkpointing, and reduced checkpointing using the LVS—we obtained similar simulation time results for the other reduced checkpoint techniques RLVS, TMI and RTMI. These three SSI techniques are considered in combination with the two most accurate sample warmup techniques, namely MRRL and MHS. These results show that MRRL in combination with fast-forwarding and full checkpointing are equally slow. The average total simulation time is more than 14 hours per benchmark. If we combine MHS with fast-forwarding, the average total simulation time per benchmark cuts down to 5 hours. This savings over MRRL is achieved by replacing warming with fast-forwarding and loading the MHS. Combining MHS with full checkpointing cuts down the total simulation time even further to slightly less than one hour. Combining the reduced checkpoint LVS approach with MHS

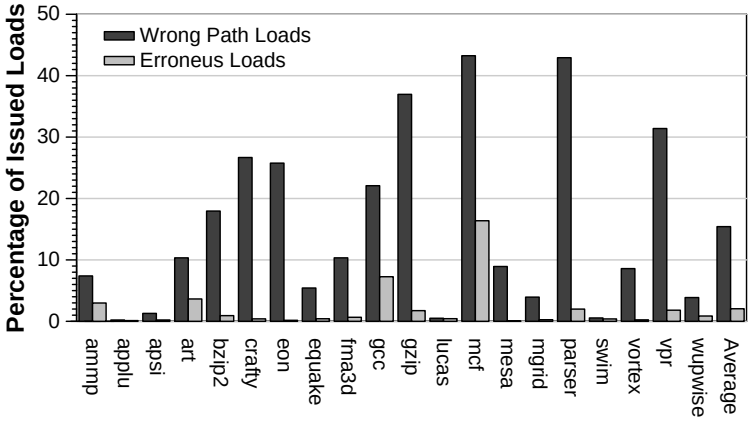

Figure III.8: Analysis of wrong-path loads while using LVS.

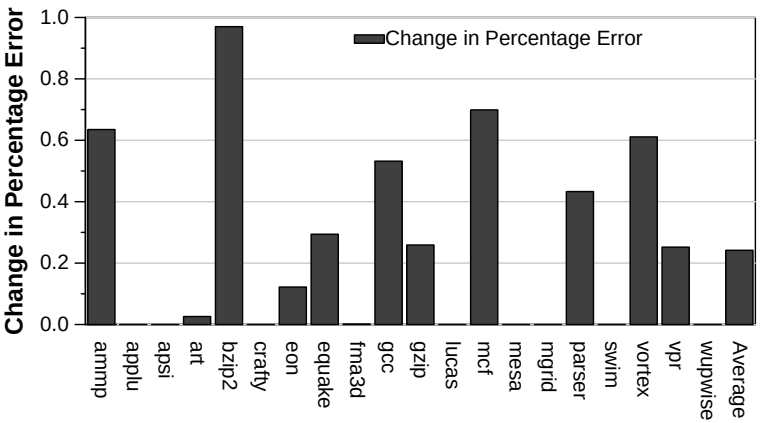

Figure III.9: Change in percentage error due to LVS.

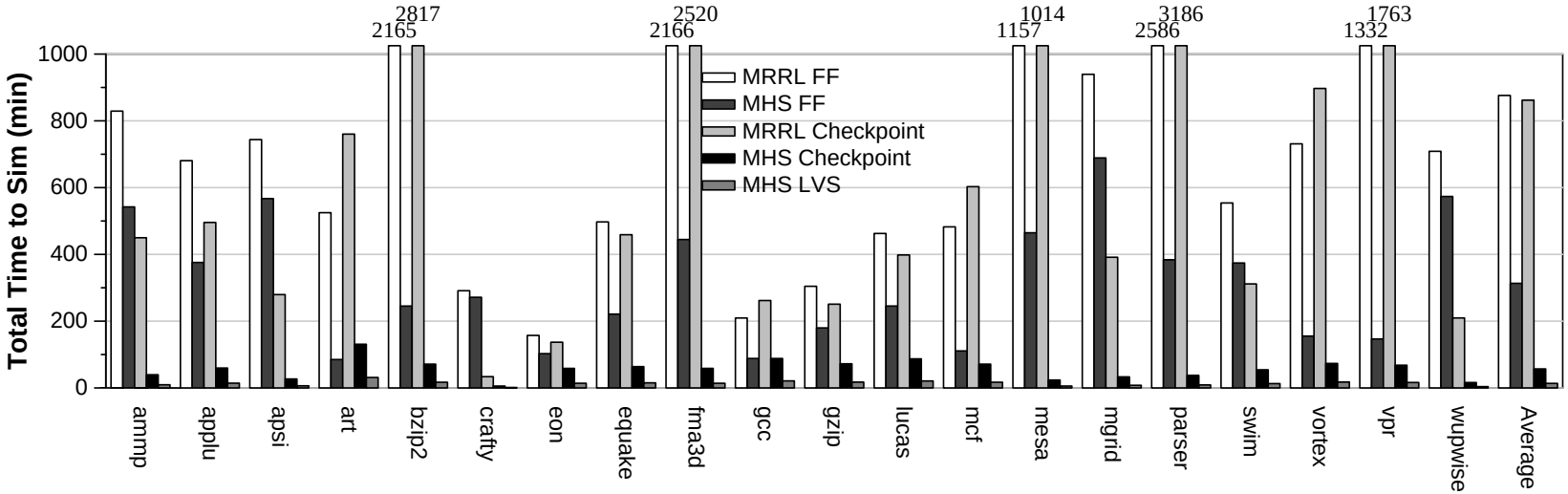

Figure III.10: Total time to simulate all samples including fast-forwarding, loading checkpoints, warming and doing detailed simulation.

reduces the average total simulation time per benchmark to 13 minutes. We obtained similar simulation time results for the other reduced checkpoint techniques.

As such, we conclude that the Sample Startup techniques proposed in this chapter achieve full detailed per-benchmark performance estimates with the same accuracy as MRRL. This is achieved in the order of minutes per benchmark which is a 63X simulation time speedup compared to MRRL in conjunction with fast-forwarding and checkpointing.

III.C.7 Storage Requirements

Figures III.11 and III.12 show the average and total sizes of the files (in MB) that need to be stored on disk per benchmark when SimPoint is used with various Sample Startup approaches: the Full Checkpoint, the Load Value Sequence (LVS), the Reduced Load Value Sequence (RLVS), the Touched Memory Image (TMI) and the Memory Hierarchy State (MHS). Clearly, the file sizes for Full Checkpoint are huge. The average file size per checkpoint is 49.3MB (see Figure III.11). The average total file size per benchmark is 7.4GB (see Figure III.12). Storing all full checkpoints for a complete benchmark can take up to 28.8GB (`lucas`). The maximum average storage requirements per checkpoint can be large as well, for example 163.6MB for `wupwise`. Loading and transferring over a network such large checkpoints can be costly in terms of simulation time as well.

The SSI techniques, namely LVS, RLVS, TMI and RTMI, result in a checkpoint reduction of more than two orders of magnitude, see Figures III.11 and III.12. The results for RTMI are similar to those for TMI. Since TMI contains at most one value per address and no zeros, size improvements can only come from situations where the first access to an address is a write and there is a later

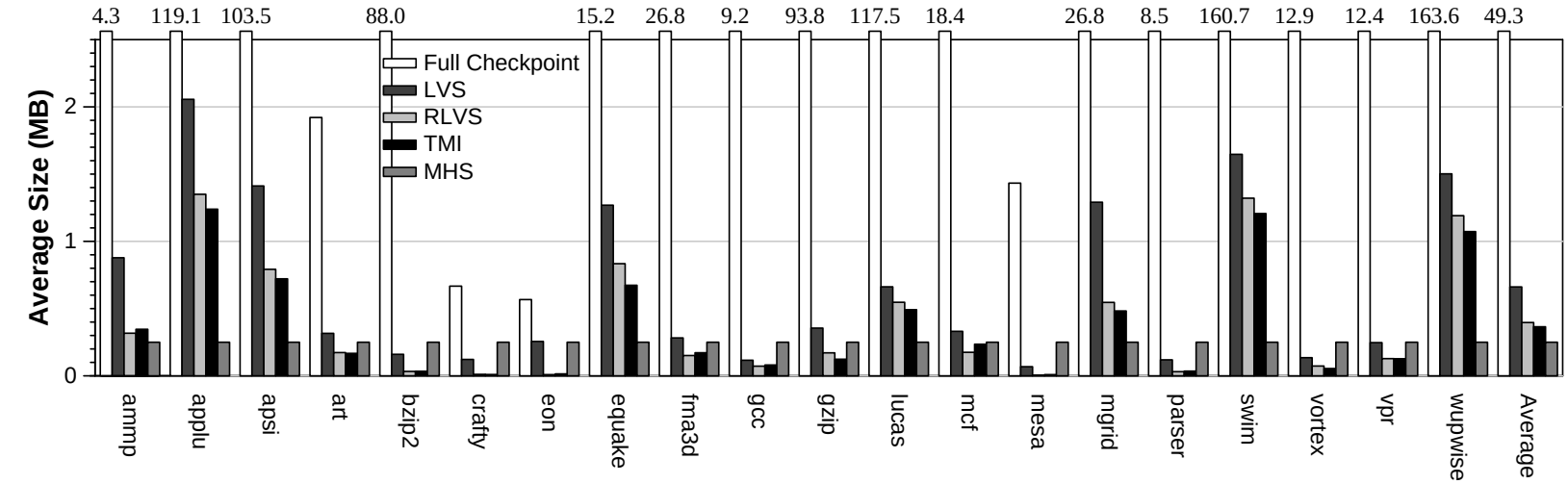

Figure III.11: Average storage requirements per sample.

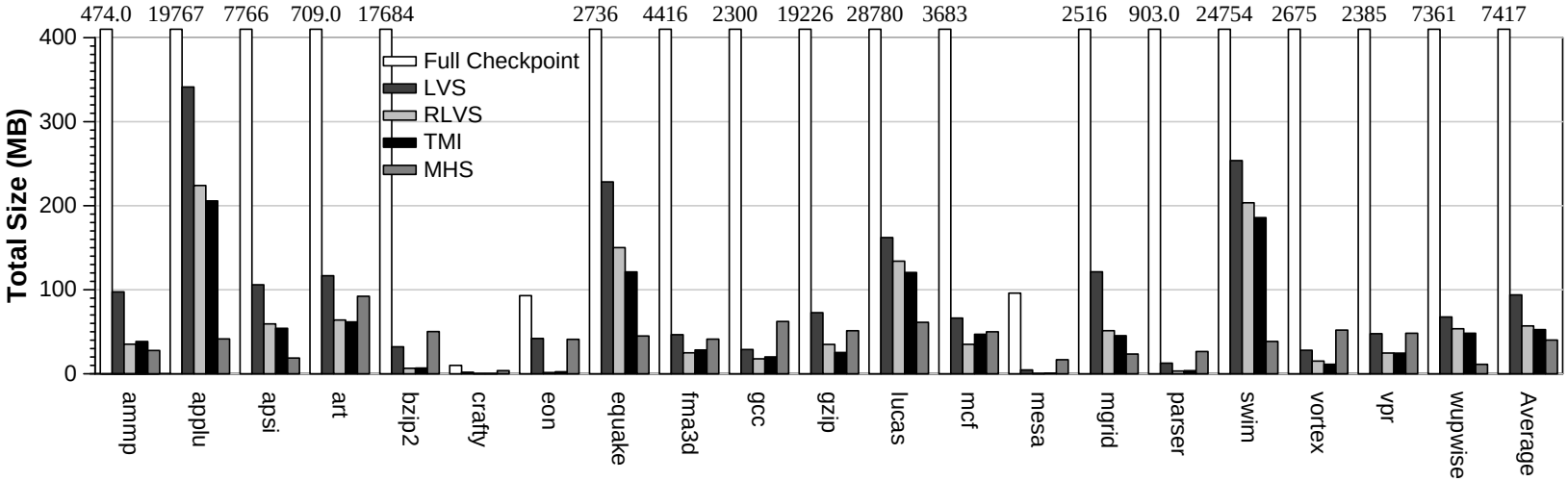

Figure III.12: Total storage requirements per benchmark.

read from that address. This is fairly rare, so the improvements are small.

The average total file sizes per benchmark for LVS, RLVS and TMI are 93.9MB, 57MB and 52.6MB, respectively; the maximum total file sizes for are 341MB, 224MB and 206MB, respectively, for `applu`. These huge checkpoint file reductions compared to full checkpoints make checkpointing feasible in terms of storage cost for sampled simulation. Also, the typical single checkpoint size is significantly reduced to 661KB, 396KB and 365KB for LVS, RLVS and TMI, respectively. This makes loading the checkpoints highly efficient.

Memory Hierarchy State (MHS) was the only warmup approach discussed that requires additional storage. Figures III.11 and III.12 quantify the additional storage needed for MHS. The total average storage needed per benchmark is 40MB. The average storage needed for MHS per checkpoint is 256kB (8 bytes per cache block). Note that this is additional storage that is needed on top of the storage needed for the checkpoints. However, it can be loaded efficiently due to its small size.

III.C.8 Using MHS and LVS with SMARTS

The SMARTS infrastructure [65] accurately estimates CPI by taking large numbers of very small samples and using optimized functional warming while fast-forwarding between samples. Typical parameters use approximately 10000 samples, each of which is 1000 instructions long and preceded by 2000 instructions of detailed processor warmup. Only 30M instructions are executed in detail, so simulation time is dominated by the cost of functional warming for tens or hundreds of billions of instructions.

We improved SMARTS' performance by replacing functional warming with our MHS and LVS techniques. Due to the very small sample length there was insufficient time for the TLB and branch predictor to warm before the end

of detailed simulation warmup. Therefore, for SMARTS we enhanced MHS to include the contents of the TLBs and branch predictor. TLB structures can be treated just like caches when considering various TLB sizes, but branch predictors need to be generated for every desired branch predictor configuration. With these changes we were able to achieve sampling errors comparable to the error rates presented in section III.C.5 for the 1M-instruction samples. In addition, the estimated CPI confidence intervals are similar to those obtained through SMARTS.

Storing the entire memory image in checkpoints for 10000 samples is infeasible, so we used LVS. Due to the small number of loads in 3000 instructions, a compressed LVS only required a single 4 kB disk block per sample. The total disk space per benchmark for the LVS checkpoint is 40 MB. Disk usage however is dominated by MHS, with total storage requirements of approximately 730 MB for each benchmark. By comparison, our SimPoint experiments used under 100 MB on average for full LVS, 50 MB for RLVS and 40 MB for MHS. In terms of disk space, SimPoint thus performs better than SMARTS.

On average, the total simulation time per benchmark for SMARTS with LVS and MHS is 130 seconds on average. About two-thirds of this time is due to decompressing the MHS information.

In contrast to the fact that the amount of disk space required is approximately 8 times larger with SMARTS, SMARTS is faster than SimPoint: 130 seconds for SMARTS versus 13 minutes for SimPoint. The reason for this is the larger of number of simulated instructions for SimPoint than for SMARTS.

Concurrently with our work, the creators of SMARTS have released TurboSMARTS [62], which takes a similar approach to the one that we have outlined here. Their documentation for the new version recommends estimating the number of samples that should be taken when collecting their version of

MHS and TMI data. The number of samples is dependent upon the program's variability, so for floating-point benchmarks this can greatly reduce the number of samples, but in other cases more samples will be required. As a result, the average disk usage is 290 MB per benchmark, but varies from 7 MB (`swim`) to 1021 MB (`vpr`). This is still over twice as large than the disk space required for SimPoint using 1-million instruction intervals.

III.D Summary

Today's computer architecture research relies heavily on detailed cycle-by-cycle simulation. Since simulating the complete execution of an industry standard benchmark can take weeks to months, several researchers have proposed sampling techniques to speed up this simulation process. Although sampling yields substantial simulation time speedups, there are two remaining bottlenecks in these sampling techniques, namely efficiently providing the sample starting image and sample architecture warmup.

This chapter proposed reduced checkpointing to obtain the sample starting image efficiently. This is done by only storing the words of memory that are to be accessed in the sample that is to be simulated, or by storing a sequence of load values as they are loaded from memory in the sample. These reduced checkpoints result in two orders of magnitude less storage than full checkpointing and faster simulation than both fast-forwarding and full checkpointing. We show that our reduced checkpointing techniques are applicable on various sampled simulation methodologies as we evaluate them for SimPoint, random sampling and SMARTS.

This chapter also compared four techniques for providing an accurate hardware state at the beginning of each sample. We conclude that architecture checkpointing and MRRL perform equally well in terms of accuracy. However,

our architecture checkpointing implementation based on the Memory Hierarchy State is substantially faster than MRRL. The end result for sampled simulation is that we obtain highly accurate per-benchmark performance estimates (only a few percent CPI prediction error) in the order of minutes, whereas previously proposed techniques required multiple hours.

III.E Acknowledgements

This chapter contains material from *Efficient Sampling Startup for Sampled Processor Simulation* [55], in *IEEE Micro Magazine*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from *Efficient Sampling Startup for SimPoint* [57], in *IEEE Micro Magazine*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper.

IV

The Co-phase Matrix

Individual programs exhibit phase behavior in which each phase has roughly uniform IPC, misprediction rates, data miss rates, and so forth. Some programs change phase rarely or in predictable ways; others such as `gcc` frequently change phase in complex ways. On a multithreaded machine, each program affects all of the others in ways determined by its current phase behavior. As a result, the combination of programs can have more complex phase behavior since it is the product of their individual behaviors. This combined behavior determines the relative progress of the threads.

This chapter focuses on improving the efficiency of multithreaded workload simulation, answering the question "Given a multi-program workload with each program starting at a specific starting point, how can we accurately and efficiently estimate performance using sampling?" We focus on an SMT processor with two hardware contexts, but also briefly examine using four hardware contexts. Our sampling approach to multithreaded simulation is guided by the phase behavior found in single-program execution. This relies on finding the phase-based behavior of each program using SimPoint [48] to classify fixed-size intervals of execution into phases. For our experiments we selected programs from the SPEC benchmark suite that show a wide variety of single program

phase-based behaviors, including many programs with complex structures.

The main contribution of this chapter is the creation of a *Co-Phase Matrix* and using it to guide the simulation of an SMT processor for a multi-program workload. The co-phase matrix represents all of the potential phase combinations of a multi-program workload to be examined in an architecture study. Our simulation approach populates the co-phase matrix with samples during simulation. Once a phase combination has an appropriate sample, we no longer need to simulate that combination and we can just fast-forward execution to the next phase combination. The amount to fast-forward is determined by the performance samples stored in the co-phase matrix. When we don't have the sample required to fast-forward execution, we can sample from the current execution offset. We also examine a method that allows us to sample all co-phase matrix entries in parallel and then, with no additional simulation, determine the IPC of the programs from any relative starting offsets.

Our work is not the first to point out that the behavior of one program running on an SMT processor can be affected by a different program that is scheduled to run at the same time. Indeed, both the work of Snavely and Tullsen [51] and of Parekh, Eggers and Levy [38] demonstrate that not only does this effect occur, but that it can in some circumstances be exploited for increased schedule efficiency. However, neither of these approaches make use of phases to perform optimizations and instead assume that each of the programs have homogeneous behavior over large time scales. We extend this idea and show that there is thread interference that happens at the level of phases, and that phases can be used to perform more accurate performance estimation.

IV.A Background

Several methods for evaluating the performance of multithreaded processors were discussed in Section II.B. Clearly, methods requiring repeated execution of entire programs are unsuitable for simulation. We use this section to explain the complexity of multithreaded execution and show why existing single- and multithreaded approaches to sampling are either inapplicable to this new domain or unlikely to get accurate results.

IV.A.1 Sampling Challenge for a Multithreaded Processor

Single-threaded sampling methods assume that sampling points can be easily determined independently from detailed simulation, either through random sampling or some heuristic. But on a multithreaded processor, the threads share the hardware resources, and it is necessary to model the co-execution of the threads to determine which instructions from the programs will be executed at the same time.

Figure IV.1 shows how IPC changes over time for each program when it is run *by itself* on the baseline SMT processor. The x-axis represents time, and the numbers on the x-axis represents the percent of execution. The y-axis is the IPC for each execution interval. The time-varying IPC behavior is shown for 10 billion instructions, after fast-forwarding for one billion instructions. In `bzip`, `gcc` and `vpr` we see periodic behavior when these programs are executed by themselves.

Figure IV.2 shows the time-varying IPC results for all two-program combinations of `bzip2`, `gcc` and `vpr` running on an SMT Processor. These figures show the IPC for each program when co-executing with the others. The per-program IPC is lower than in Figure IV.1 because now all the programs are fighting for the resources they once had to themselves. The first thing to note is

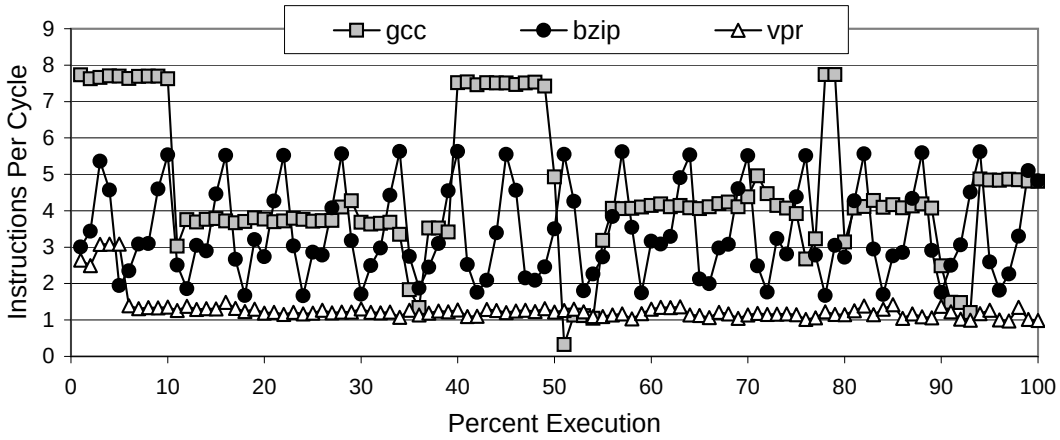

Figure IV.1: IPC Time-varying behavior for each program when it is run by itself on the SMT processor. The x-axis scale is percentage of execution.

that there is significantly more complex behavior here than was present when the programs were running as individuals. Upon further inspection, patterns begin to take shape, showing synchronized changes in behavior.

One result that stands out is that the phased-based behavior seen during these SMT runs is dominated mainly by `bzip` in its combinations. The program `vpr` provides an interesting baseline for all of these figures as its behavior remains fairly stable over time. When you compare that behavior to the behavior that it shows when paired with a program such as `gcc`, you can clearly see that the addition of multiple threads greatly complicates the task of estimating the performance of any one thread, even if that one thread by itself is not that complex. Indeed, while sometimes the IPC of one program will go up while the other goes down, at other times they both go up or down together. Invariably, though, the changes are due to the the existing phase-behavior of one program or the other.

As can be seen in Figure IV.2, the task of determining the relative execution rates for several programs will be complex. For a single pair of programs, such as `gcc` and `bzip2`, at times the programs will execute at the same rate

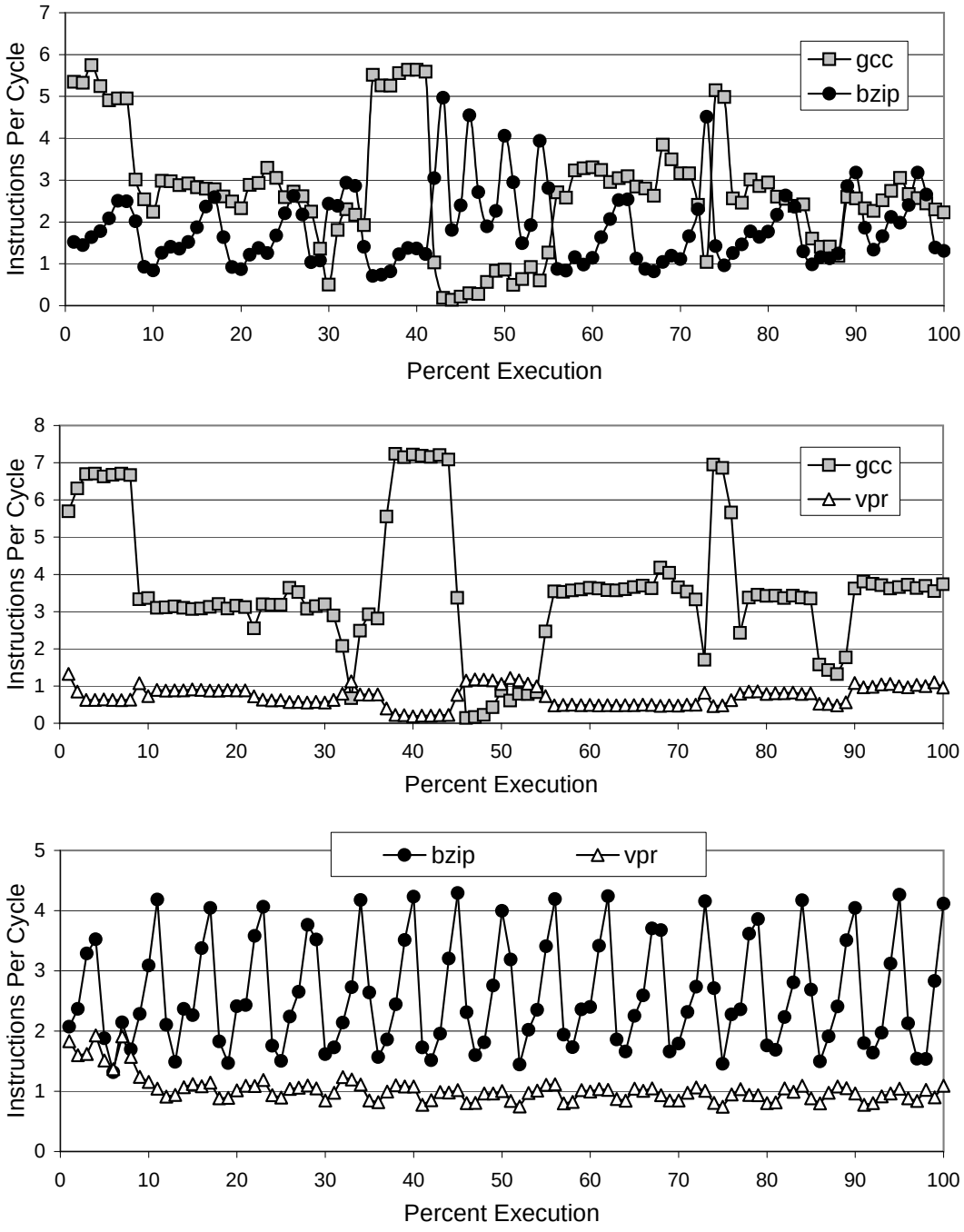

Figure IV.2: Time Varying IPC when running all the above 2 program combinations at the same time together on a dual hardware context SMT Processor.

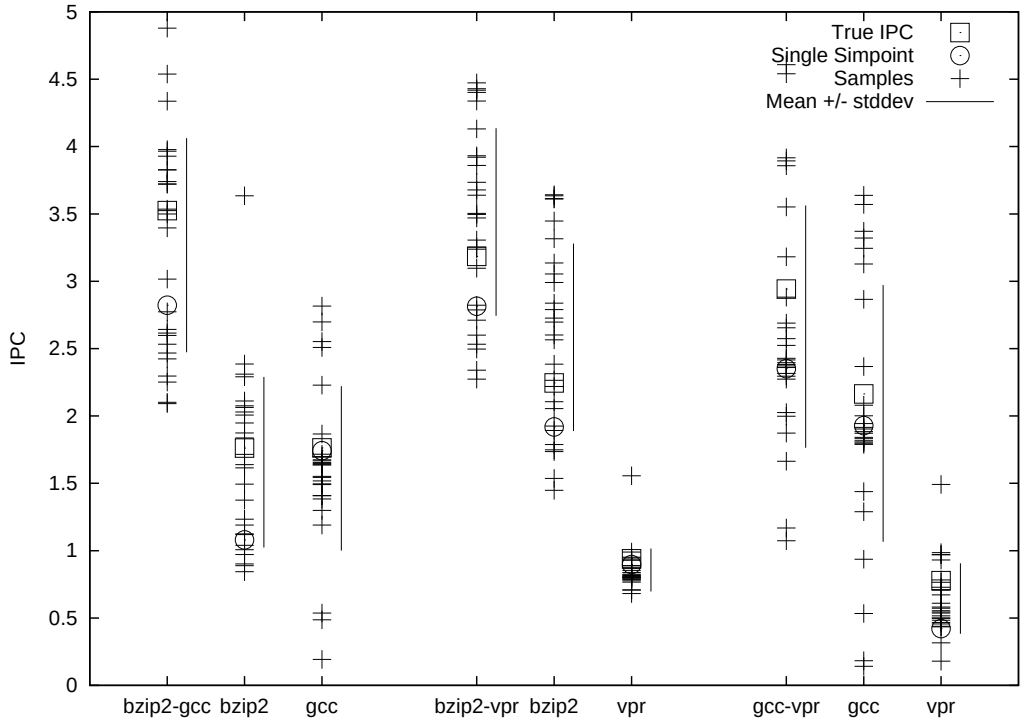

Figure IV.3: Random sampling results.

and at others there is an order of magnitude difference in IPC. It is therefore a challenge to determine between samples how much to fast-forward each separate thread in order to arrive at a real sample that would exist in the representative baseline full simulation.

Capturing the complex behavior seen in these time-varying results is beyond is beyond the capability of existing methodologies which rely on the execution of just one interval from each program. Instead, an approach that takes advantage of the phase behavior of each program is needed to capture this behavior.

IV.A.2 Using Single Simulation Points

A typical sampled simulation methodology that researchers use is to simulate an SMT processor for 100 to 300 million instructions at a single point for a given multi-program workload. This only exercises the interaction between the programs for a couple of different behavior combinations (derived from sequentially occurring phases) for each program. To illustrate the problem, we first examine randomly picking an offset in each program to start executing at and measure performance. Each offset combination was simulated until a total of 300 million instructions were executed. This is shown in Figure IV.3, where IPCs for random simulations are shown for the program pairs `bzip-gcc`, `bzip-vpr`, and `gcc-vpr`; the per thread estimated IPC results are shown next to each program combination. The random sampling results are shown as plus (+) signs. We also show the overall and per-program IPCs found when executing the program combination for 10 billion instruction, and the 300M-instruction samples were drawn from this 10 billion instruction execution. The results show that arbitrarily taking one sample for a program combination can lead to highly variable IPC estimates.

Also shown in Figure IV.3 are results that use the best *single* simulation point found by the SimPoint algorithm in [48] for each program and co-simulated those together for 300 million instructions. For a given program/input pair, SimPoint profiles the code usage, broken down into 100 million samples, over the complete execution of the program. It then compares a code profile of the complete execution of the program with the profile from each interval and attempts to pick the more representative set of 100 million contiguous instructions. The circle on the graphs represents using the best single SimPoint for each program when performing an SMT simulation. The results show that for some combinations it has a relatively small error, while for others it can have an overall error

of almost 20% (`gcc-gzip`) or even almost 40% (`bzip` in `bzip2-gcc`).

Single simulation points can be fairly representative of the entire program execution. They typically capture the transition point between the two most dominant phases in the run. Vera *et al.* [60, 61] refined this technique by repeatedly executing the 300M-instruction intervals to reflect the possible phase interactions within them, ultimately executing billions of instructions per program pair. Nonetheless, most programs have many phases and the single simulation point will have a higher error rate than if samples are taken from each phase of the program's execution. Therefore, we focus on obtaining a more accurate picture of the program's execution by taking samples from all of the phase combinations seen for a multi-program workload.

IV.B Discussion

In this section we develop the co-phase method in several variations and evaluate them against each other and simpler techniques. At the end, we revisit one of them, the static co-phase method, using more refinements for increased efficiency and accuracy as well as an improved evaluation methodology.

IV.B.1 Guiding Fast-forwarding Using the Last Sample

Figure IV.2 shows complex fine-grain phase behavior, but general trends of execution are often present for significant intervals. This is caused by the phase nature of programs, where programs tend to execute in the same phase for a given period of time before transitioning to a new phase [2, 49]. Therefore the performance from recent execution can be a reasonable prediction of near future performance.

Using this observation, as a baseline sampling technique we examine a straight-forward sampling approach where we assume that the program's execu-

tion will continue to have the same IPC for some time. Then performance from one sample can be used to guide the amount of per-thread fast-forwarding until the next sample occurs. By periodically resampling we can detect changes in per-thread behavior, and then correct the fast-forwarding until the next sampling.

Longer samples allow us to ignore high-frequency variation, concentrating on general performance trends. (A short sample may give results that are not as representative a longer one.) Shorter resampling periods allow us to detect brief performance variations, increasing accuracy at the expense of more samples. If the number of samples is large, the samples may need to be reduced in length to keep simulation time reasonable.

IV.B.2 Finding Phases to Improve Sampling

The technique discussed in the last section, periodically sampling and then assuming the behavior will be stable until the next sample, is simple to implement and works reasonably well in many cases. However, with only an incremental amount of complexity, and leveraging existing work in phase analysis, we can do even better. This new approach anticipates phase changes independently from sampling, allowing samples to be taken at every new phase combination. Figure IV.1 shows the repetitive phase behavior of single threads, and even when two-program combinations are run in Figure IV.2. The per-thread phase behavior is still present in multithreaded execution but the phases are now affected by competition for resources from other threads. This leads us to propose the use of phase detection techniques based on phases discovered using our earlier work on single-threaded program analysis.

For our approach, the phase behavior is represented by a *phase-ID trace* representing the complete execution of a single program, where each phase is represented by a unique ID determined by the SimPoint program. Section II.A

explained how SimPoint is used to analyze a program, identifying phases and representative simulation points. The phase-ID trace indicates at which instructions in the program's execution phase changes occur and the new phase IDs.

IV.B.3 The Co-Phase Matrix

On a multithreaded processor, the state of each thread's execution can be represented by the per-program phase-ID it is currently executing in. The key idea of our technique is that, just as in the single threaded version the overall behavior does not change within a given phase, in a multithreaded machine the overall behavior should not change unless *at least one* thread has a phase change. Thus we need to keep track of a list of all *combinations* of phases that have been seen running together. This combination of phase-IDs, which are executing together, represent a *unique co-phase identifier*. We have found that taking a sample of the simultaneous execution of programs and storing it with its co-phase identifier accurately represents the multithreaded performance when this same co-phase combination is seen again in the future. We can then store a list of the past combinations we have seen, and we term this list the *co-phase matrix*.

A co-phase matrix represents the combination of all of the phase-IDs from each program in the workload that can execute simultaneously on the multithreaded machine, where there is an entry in the matrix for each co-phase identifier. If each phase combination were simulated, then the table would be filled with representative samples for all the possible phase combinations in the program. For each combination of the phases (*co-phase*) that occurs when co-executing two or more programs we store into the co-phase matrix the per-thread IPC found during a sample of detailed simulation.

IV.B.4 Guiding Fast-Forwarding

We can estimate the co-execution of multiple programs at a given point in time if we have two items: the phase-ID trace and the co-phase matrix entry that corresponds to the phases currently executing. If we have the entry in the co-phase matrix then we can predict per-thread IPC because we have observed and recorded this set of phases executing in the past. From SimPoint we obtain the phase-ID trace, which is used to determine how many instructions each program must execute before it encounters the next phase change. Using the IPC estimates from the co-phase matrix and the phase-ID trace we determine how many cycles it would take for the next phase change to occur for any of the co-executing threads. This is used to guide how far to fast-forward each thread using the estimated IPCs from the co-phase matrix. The following is an overview of the algorithm:

1. Co-Phase Matrix Lookup - The current co-execution thread combination represents a co-phase identifier, which is looked up in the co-phase matrix. If a sample exists, retrieve the per-thread IPC for each thread. If a sample does not exist, then perform detailed simulation for a specified sample size (see Section IV.B.7), and store the per-thread IPC into the co-phase matrix to reuse later.

2. Determine Number of Cycles to Fast-Forward - Using each program's phase-ID trace, calculate the number of instructions until the next phase change for each thread. Use this and the per-thread IPC from the co-phase matrix to calculate the number of cycles to reach that phase change. The thread with the smallest number of cycles until the next phase change determines how far to fast-forward.

3. Fast-Forward to Next Phase Change - Take the number of cycles from step 2,

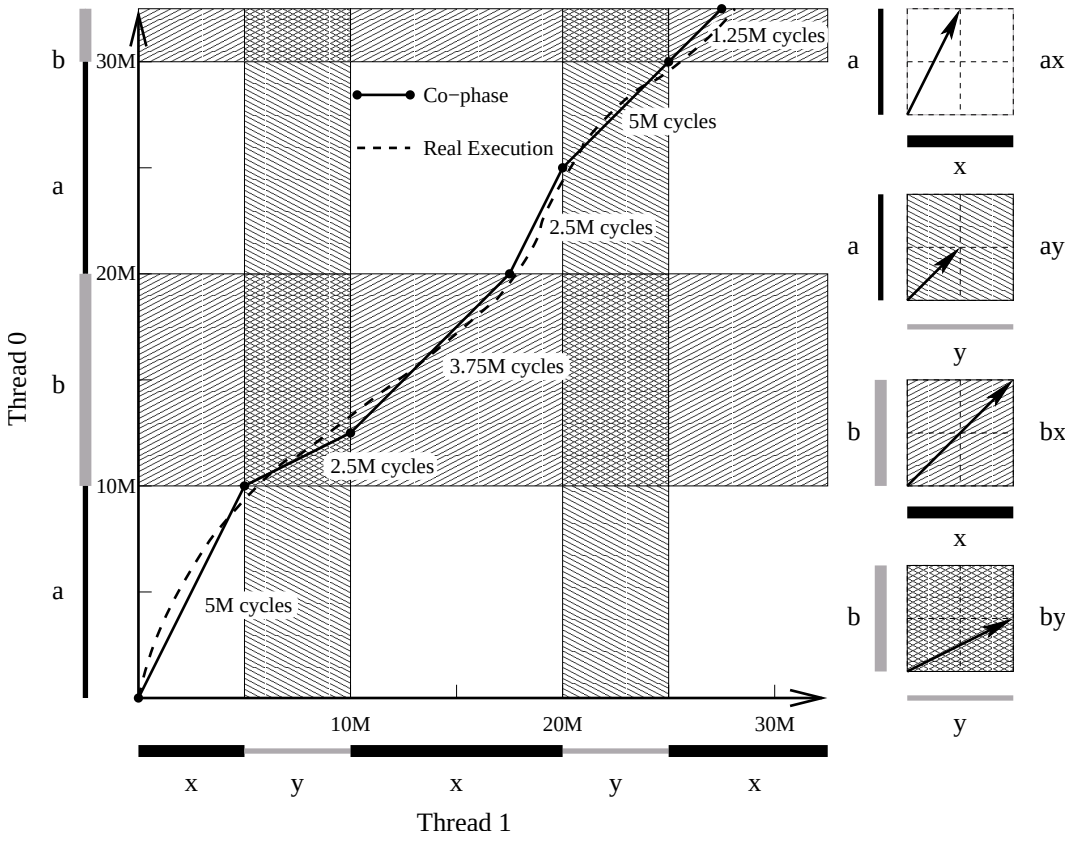

Figure IV.4: Approximating detailed execution with the co-phase matrix.

and multiply this by the per-thread IPC from the co-phase matrix to determine how many instructions to fast-forward each thread. Fast-forward each thread that many instructions. This results in a new phase-combination, and go back to step 1.

A synthetic example of using this approach can be found numerically in Table IV.1 and graphically in Figure IV.4. The top table shows the per-thread phase-ID trace. The middle table shows the resulting co-phase matrix built up using detailed samples during simulation. Finally, the bottom table shows the results of using the co-phase method.

Table IV.1: Phases found in two programs (5M instruction intervals) and a co-phase matrix. The table on the top shows the phase-ID trace gathered from SimPoint. The matrix in the middle shows an example final co-phase matrix from simulating the two threads together. The bottom table shows the results of co-phase matrix simulation.

Instructions (M)	Thread 0 Phase	Thread 1 Phase
0–5	a	x
5–10	a	y
10–15	b	x
15–20	b	x
20–25	a	y
25–30	a	x
30–35	b	x

Co-Phase	Thread 0 IPC	Thread 1 IPC
ax	2	1
ay	1	1
bx	2	2
by	1	2

Thread 0 Inst (M)	Thread 1 Inst (M)	Co-Phase	Cycles (M)
0–10	0–5	ax	5
10–12.5	5–10	by	2.5
12.5–20	10–17.5	bx	3.75
20–25	17.5–20	ax	2.5
25–30	20–25	ay	5
30–32.5	25–27.5	ay	1.25

In Figure IV.4, the dashed line on the graph represents the relative progress of the two threads during a hypothetical real execution while the solid line is the approximation made using the co-phase method. The phase-ID traces of the two threads (thread 0 on the y-axis and thread 1 on the x-axis) can be seen adjacent to the axes. Thread 0 goes through the phase sequence abab with phase changes at 10M, 20M and 30M instructions, respectively; thread 1 goes through the phase sequence xyxyx with phase changes at 5M, 10M, 20M and 25M instructions, respectively. Each rectangular section of the graph is shaded according to the co-phase that will be used while the execution of the two threads is within the rectangle. For example, the white-shaded rectangles represent the co-phase ax, where phase a for thread 0 is co-executing with phase x from thread 1; the heaviest shaded rectangles represent the co-phase by, *etc.* The graphs on the right of the relative progress graph visually represent the co-phase matrix. For each co-phase we show a vector indicating the relative progress of the threads. For co-phase ax, the co-phase matrix shows that phase a in thread 0 makes twice as fast progress as phase x in thread 1. For by, phase y makes twice as fast progress as phase b. For co-phases ay and bx, both phases make both equally fast progress, but ay has half the throughput of co-phase bx.

For this example, we assume that the execution begins at $(0, 0)$, the start of both programs, but we could apply the same procedure for any other starting point. In the example, the starting point is in co-phase ax. From the co-phase matrix entry corresponding to ax, we see that thread 0 progresses twice as fast as thread 1. Execution continues at that rate until both programs change phases at $(5M, 10M)$. Thus it takes 5M cycles to exit the first co-phase. The new co-phase is by with IPCs of 2 and 1 for threads 0 and 1, respectively. After just 2.5M cycles the horizontal thread leaves phase y at point $(10M, 12.5M)$. Progress beyond this point will be in co-phase bx. Both threads will now make

equal progress, *i.e.*, we assume both threads run at an IPC of 2, until one of the thread hits a new phase ID (phase change). The next co-phase change will occur at point $(17.5M, 20M)$ after 3.75M cycles. The new co-phase then is ax, *etc.* This process is repeated until our target execution length is achieved, closely following the real execution of the two threads.

Given a co-phase matrix, the starting points for two threads, and the phase-ID trace, we quickly estimate the overall IPC for the pair of programs running on a multithreaded processor using the above approach. Next we examine two different approaches for guiding simulation using the contents of the co-phase matrix.

IV.B.5 Estimating Performance with a Dynamic Co-Phase Matrix

The first approach we propose for guiding simulation dynamically populates the co-phase matrix as we simulate. For this technique, we start a simulation from a desired initial offset of each program with an empty co-phase matrix. The start of execution represents the first co-phase matrix entry needed to execute, so we perform a detailed simulation of that co-phase at the current execution and fill in the corresponding co-phase matrix entry. From this matrix entry we estimate how many instructions are going to execute from each program before the next phase change, using knowledge of how many instructions we have executed from each individual program and the single program phase-ID trace. We then fast-forward each program thread by that many instructions, and examine the new co-phase that the workload is at. If this co-phase is not in our co-phase matrix (or, for example, is based on too few samples), we perform detailed multithreaded simulation and fill it in. If it is in our matrix, we use the existing entry to estimate the number of instructions executed for each program and fast-forward each program. This process is repeated until we have completed simulation. When we

are done we have a co-phase matrix filled out with all of the co-phases that were observed during this workload's execution, and a weight is assigned to each co-phase matrix entry corresponding to the fraction of time each co-phase occurred during this process. The performance results gathered in this co-phase matrix are then combined to achieve an overall estimated IPC of the combined run, and a per thread IPC.

The results labeled First Phase, 1% Phase and 5% Phase use this dynamic co-phase matrix approach. First Phase uses only the first sample found. The results labeled 1% and 5% add new samples at regular intervals. To sample 5% of execution, for example, a new sample is taken every 20 times (5 out of 100) that a specific co-phase matrix entry occurs during simulation.

IV.B.6 Estimating Performance with a Static Co-Phase Matrix

We now describe using a static co-phase matrix to guide simulation. For each co-phase that could occur in the matrix we simulate the programs together at representative simulation points. We used the SimPoint algorithm to find the representative simulation points from each phase. The simulation points might not actually co-execute during the baseline comparison we are trying to model, but they embody an average behavior for that co-phase that makes them representative. For this approach, only a single sample is used for each co-phase matrix entry from running the SimPoint simulation point combinations together.

To arrive at an overall IPC, the multi-program workload will start its execution in one of the co-phase matrix entries. Using that co-phase matrix entry and the individual program phase-ID trace information, we predict how many cycles until the next phase change and the number of instructions to be executed from each program. We then advance to a new co-phase simulation matrix entry and advance each program by its number of estimated instructions executed, and

we repeat this process. Once we reach the stopping criteria for our simulation (*e.g.*, reaching a total number of simulated instructions or a minimum number from each thread) we have the total number of cycles to execute the workload and the number of instructions executed per thread. Note, this analytical simulation is done *after* we have the co-phase matrix. It requires neither functional nor detailed simulation after the static matrix has been created. At this point the analytical simulation is completely driven by the co-phase matrix.

This method is extremely efficient when used with checkpoints. One set of checkpoints suffices for each combination of programs, and can be used for every architectural change to be examined. For a particular architecture configuration, after simulating each phase combination once for just tens of millions of instructions, we can then arrive at the estimated performance results for all possible thread starting offsets. With checkpoints, these simulations may be done in parallel if there are sufficient resources. Unfortunately, if the number of threads is large or they have complicated phase behavior, there will be too many potential co-phase matrix entries to simulate. In this case, the dynamic approach described above for filling in the co-phase matrix to guide sampling and fast-forwarding would be preferred. Alternately, the static co-phase matrix can be collected as each entry is needed during simulation and the partial static co-phase matrix can be shared and updated by several simulations concurrently.

IV.B.7 Original Methodology

For this research we focus on eight programs to create a representative workload from the different types of phase behavior we saw in our prior SimPoint research. We use `bzip2`, `equake`, `gcc`, `lucas`, `gzip`, `mesa`, `perl` and `vpr` to examine multi-program phase-based interactions. The first four programs represent the most complicated phase-based behavior found in the SPEC benchmark suite,

Table IV.2: SMT processor configuration.

I-Cache	64kB 2-way set-associative, 64-byte blocks, 1-cycle latency
D-Cache	64kB 2-way set-associative, 64-byte blocks, 3-cycle latency
Unified L2	1 MB 4-way set-associative, 64-byte blocks, 10-cycle latency
Memory	100-cycle latency
Branch Pred	21264-style hybrid predictor with 13-bit global history indexing a 8k-entry global PHT and 8k-entry choice table; 2k 11-bit local history entries indexing a 2k-entry local PHT
OOO Issue	Out-of-order issue, 256-entry re-order buffer
Width	8 instructions per cycle (Fetch, Decode, Issue and Commit)
Func Units	6 Integer, 2 Integer Multiply, 4 FP Add, 2 FP Multiply

and the last four programs the average case phase-based behavior. We examine running all 28 combinations of the above eight programs in pairs on a two-context SMT processor. Four groups of four threads are also run on the same processor configuration, extended to four contexts.

For our multi-program workloads we fast-forwarded each program 1 billion instructions and then started co-simulating them on the SMT processor using the ICOUNT fetching heuristic [54]. We terminate simulation of a given multi-program workload as soon as the first program finishes executing 10 billion instructions. We use this to denote the end of the multi-program simulation due to the fact that each SPEC program has a different number of instructions for its reference input run, and each program executes instructions at different rates. In addition, we verified that the 10 billion instruction section for this multi-program workload is fairly representative of the phases seen over the whole program execution. This also gives us an arbitrary point part way through execution to start each program.

The M5 SMT simulator [4] from Michigan, based on SimpleScalar3.0c [5], was used to collect performance and architecture metrics in the simultaneously multithreaded environment. Although the simulator is capable of full-system simulation we did not use that capability for this work. The configuration for

this simulator is shown in Table V.B.11. It is configured to support an intensive multithreaded workload. Hence the large cache and abundant reservation stations. We simulated SPEC 2000 benchmarks compiled for the Alpha ISA. The binaries we used in this study and how they were compiled can be found at `http://www.simplescalar.com/`.

All of our results are compared against complete detailed simulation of the 10 billion instruction interval just described. Each workload is labeled with thread 0 first and thread 1 second, so thread 0 in `gcc-vpr` is `gcc`, and thread 1 is `vpr`. We examine both the overall and per-thread IPC. Although the overall IPC is of importance, it is also critical that the per-thread performance be accurate so that our simulation model can be used to study throughput, fairness, perceived user time and scheduling. Additionally, if a simulation method weights two executing threads differently than would occur in practice, then it may be effectively simulating a different workload than would occur naturally.

Phase Selection

For each program to be run in the multi-program workloads, we gathered the Basic Block Vectors for that program and identified the phases and simulation points as described in [48]. To ensure that fine-grained behavior would be evident we used 10 million instruction intervals. We used the SimPoint tools to find up to 20 phases (max K was set to 20) in each program. In Figure IV.5, we show the number of phases that were actually found for each program.

During the detailed simulation of a two-program workload, in the worst case, the number of possible co-phases between the two different programs is the product of the number of simulation points for each program. For example, in Figure IV.5 we see that `gcc` had 12 phases and `vpr` had 7, so the total possible number of co-phases is 84. Figure IV.6 shows the maximum possible number of

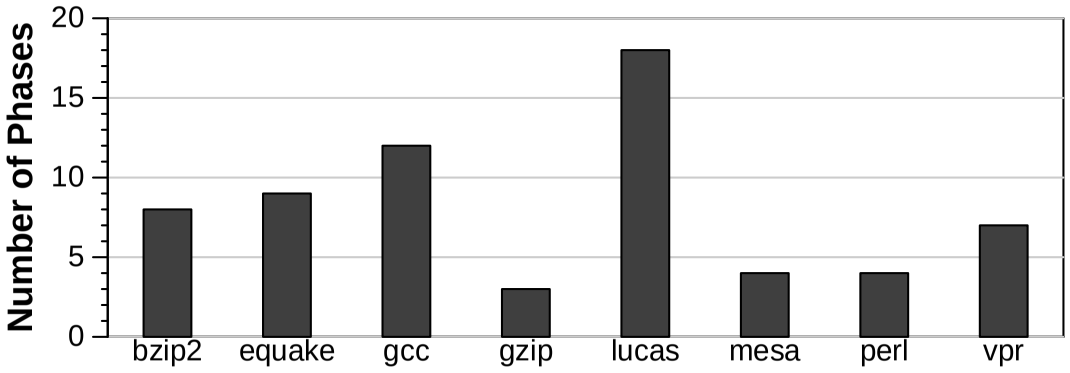

Figure IV.5: Number of phases found for each program.

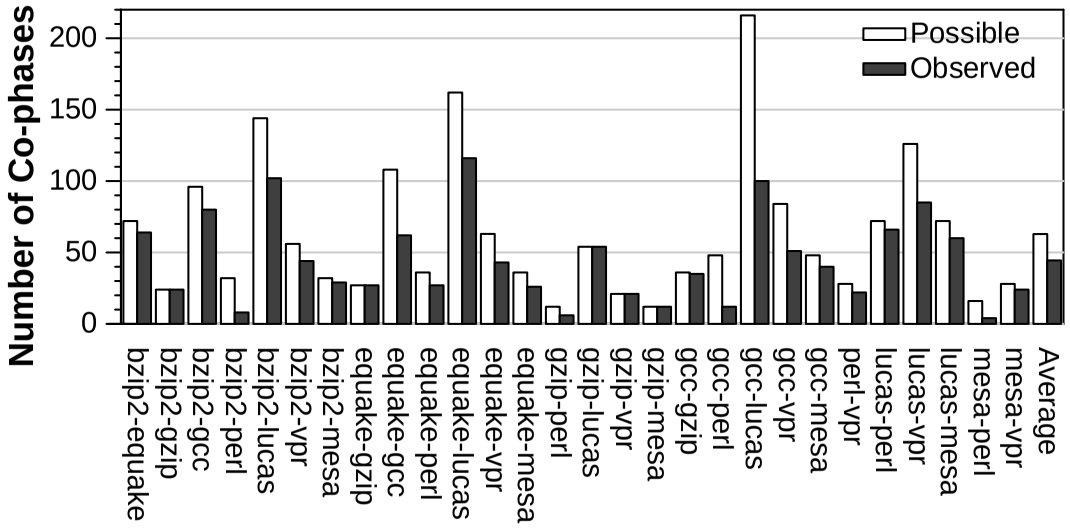

Figure IV.6: Number of phase combinations that could have occurred and the number that actually occurred during detailed simulation.

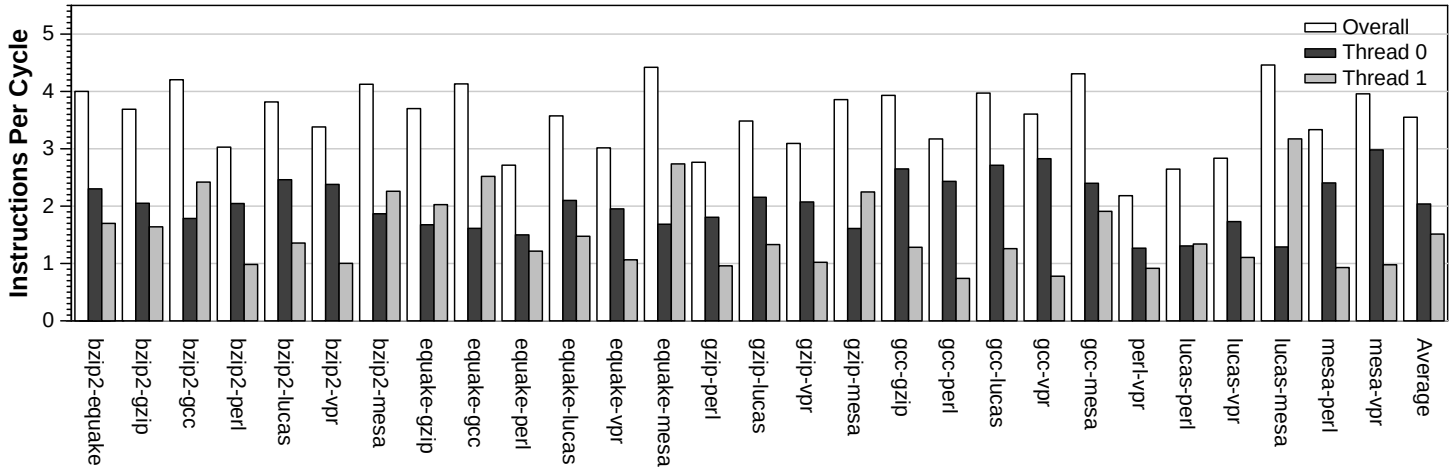

Figure IV.7: IPC statistics for all two-program combinations.

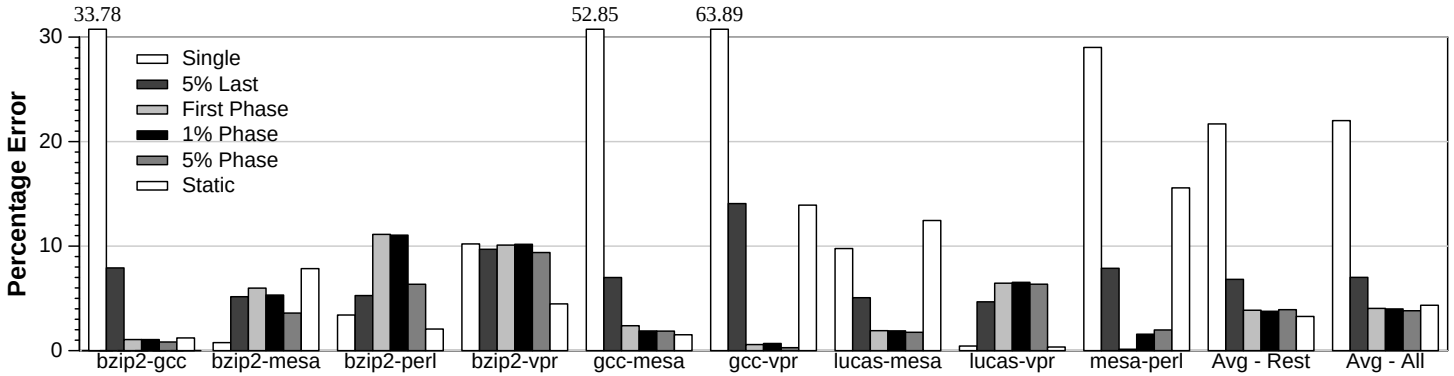

Figure IV.8: Overall IPC error comparing the different sampling techniques.

co-phase identifiers for each two-program combination we examined, as well as the number observed during simulation. Even though `gcc-vpr` had 84 possible co-phases, only 51 of the possible pairings occurred during the baseline simulation we performed. Figure IV.7 shows that the overall and per-thread IPC of each of the program combination. The pairs varied greatly in performance, with overall IPC between 2.2 and 4.5 and single-threaded performance between 0.7 and 3.2. Depending on its partners, a program's IPC can vary by as much as 1.3.

It is important to note that our fixed offsetting exercises most of the possible co-phases for most pairs of programs. A notable exception is `gcc-lucas`, which uses just under half of the possible co-phases because both programs have complex phase behavior leading to the largest number of potential co-phases. The other interesting case is `perl` which has a simple phase structure and a low IPC in combination with some other programs; `perl` does not always leave its first phase. Reaching most co-phases ensures that our tests will encounter the same range of circumstances as they would have had we run the pairs from many different offsets. If only a small fraction of phases occurred in each run (possibly because there were phases that only occurred at the start or end of thread execution) then we would have needed to run more tests. Our tests involve program combinations with significantly varying numbers of co-phases, so the full range of test complexity is covered.

Sample Collection

For each sample gathered during SMT simulation we used Hit on Cold warmup to minimize the effects of sample startup on relative thread progress. We considered three sampling methods. In these sampling techniques, a sample stops when one of the conditions below is met:

Total 5M: total 5M instructions are committed,

First 5M: a single thread commits 5M instructions, or

Both 5M: both threads commit at least 5M instructions.

The Total 5M method performed worst because it executes the fewest instructions, making warmup to great a fraction of the sample. There is a similar problem with First 5M, which may be dominated by the best-performing thread. In cases where one thread significantly outperforms the other this could lead to unbalanced sampling. Both 5M was the overall best method with only `gcc`'s frequent phase changes as notable weakness, so in our dynamic co-phase matrix experiments we sample long enough that each thread executes at least 5M instructions during the sampling. When populating a co-phase matrix entry, it is not always possible to sample that long before hitting a co-phase change. When this occurs, we take additional samples when the co-phase re-occurs and combine them with the earlier sample results. This avoids the problems with frequent phase changes. With resampling techniques, the additional samples do not need to contain 5M instructions for each thread.

The experiments with static co-phases matrices do not have the advantage of sampling from multiple locations, so they use Both 10M. The faster thread can overrun the end of the co-phase but longer samples seem to be an advantage overall. Experiments with many different ways to terminate a sample (including all of the 5M methods just mentioned) demonstrated that no technique is best for all pairs of programs.

IV.B.8 Pairwise Simulation Results

We first examine the error in IPC seen using the dynamic and static co-phase matrix simulation approaches. Figure IV.9 shows the error in IPC when guiding simulation dynamically building up the co-phase matrix using 1% sampling as described in Section IV.B.5. Figure IV.10 shows the error in IPC

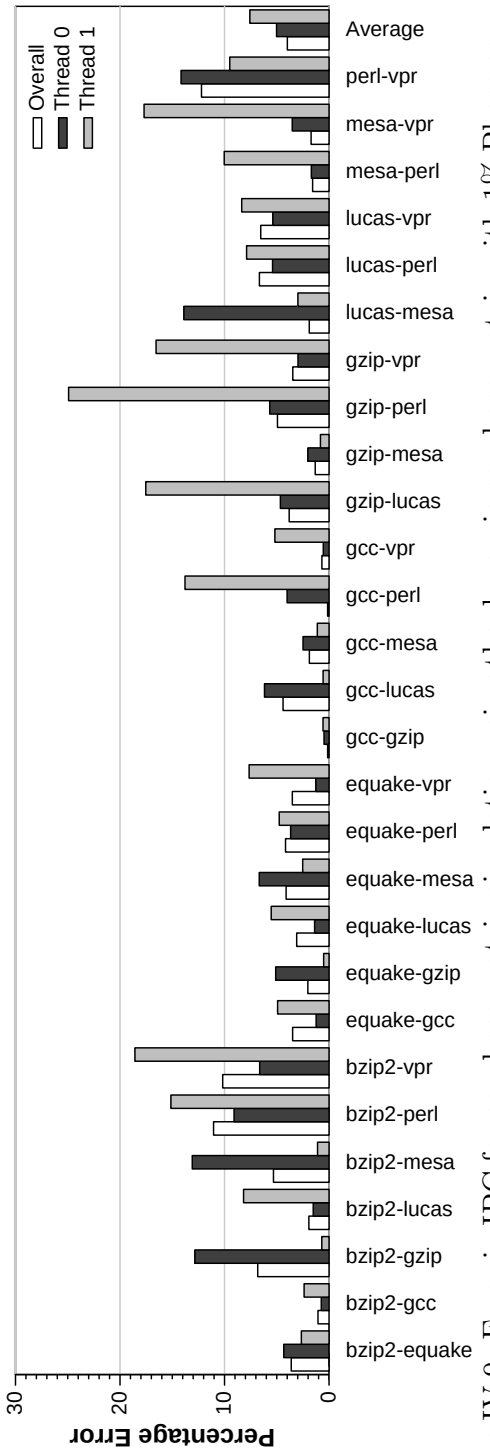

Figure IV.9: Error in IPC for co-phase matrix simulation using the dynamic co-phase matrix with 1% Phase sampling.

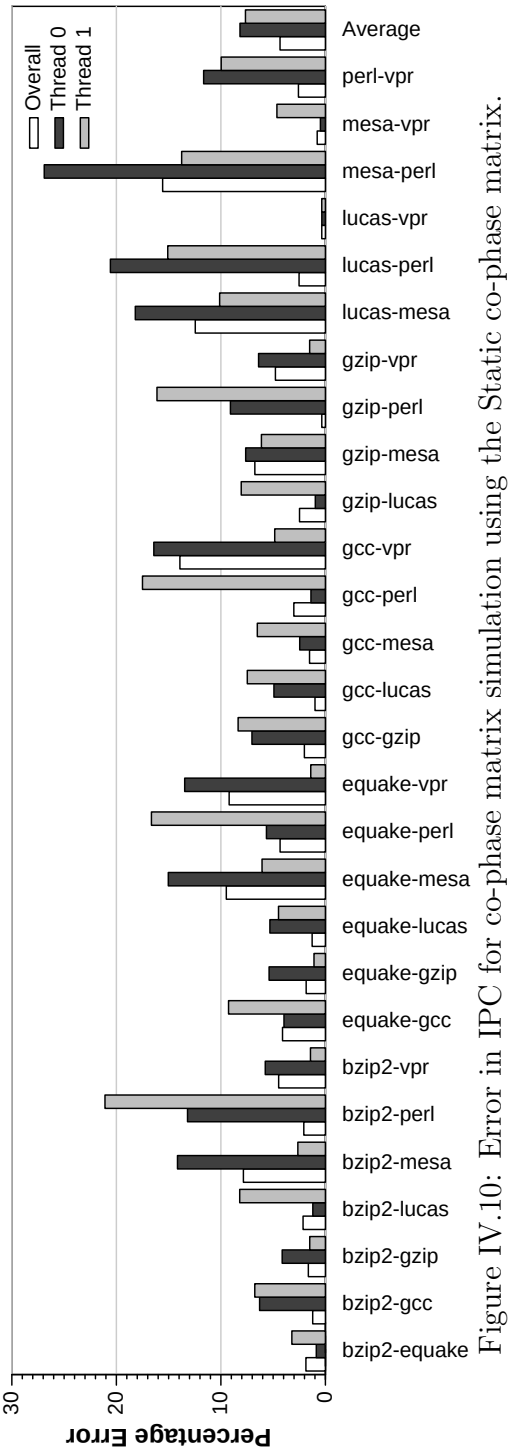

Figure IV.10: Error in IPC for co-phase matrix simulation using the Static co-phase matrix.

when using the static co-phase matrix to guide simulation. Results are shown for overall IPC error and per-thread error. Overall, for both techniques, the average error rate was 4% and the average per-thread error rate was below 8%. In the worst case (`mesa-perl`), the overall error was 16%, and the per-thread error was 27%.

Figure IV.8 compares our dynamic and static co-phase matrix approaches with alternate sampling techniques described earlier, concentrating on overall error in IPC. The first bar shows the result of co-simulating the best Single simulation points until one thread executes 100M instructions. 5% Last shows the estimated performance, where we regularly sample the per-thread IPC to predict how far to fast-forward each thread to the next sample. Each sample simulates until both threads execute at least 5M instructions. 5% of the workload has detailed simulation performed on it, and the rest of execution is skipped via fast-forwarding.

The next bar, First Phase, shows our dynamic co-phase matrix approach, where sampling is done for a co-phase matrix entry until a total of 5M instructions are simulated for each thread. This represents taking the 1st sample for each co-phase matrix entry, and using that to guide fast-forwarding for that entry for the rest of the simulation. The N% Phase methods resample each co-phase, where detailed simulation is performed for 1% or 5% of the workload's execution, and the rest is fast-forwarding. For this approach new samples are combined with the old ones. Finally, Static uses the static co-phase matrix.

The Single SimPoint method produced errors much greater than would be expected when using Single SimPoint in the single-threaded case. Even though the intervals are representative of the overall instruction mix, the pairing of single simulation points cannot capture the complexity of ongoing phase interactions. The high single-thread errors show that pairings can be quite atypical. In terms

of average error rate, phase-based sampling techniques are significantly better than the Single and 5% Last techniques.

The Static method does poorly in three notable cases: `gcc-vpr`, `lucas-mesa` and `mesa-perl`. In the first two cases programs with many frequently changing phases were combined with programs with few phases. In the latter case `perl` spends all of its time in one phase due to its simple phase structure and low IPC. The frequency of phase changes in one program makes each co-phase short and the dynamic co-phases may not contain all of the interactions that the longer co-phase sample used by the static method observes. Despite these problem programs the overall error rate is comparable to the other phase-based techniques.

'First' and '1% Phase Sample FF' use significantly fewer samples than '5% Phase Sample FF'; the decreased execution time may make them more appropriate choices than the slightly better-performing '5% Phase Sample FF'. These two methods have similar execution time because few co-phases occur often enough to be resampled at the 1% rate. Increasing the sampling rate to 5% is rarely necessary, but it does significantly help `bzip2-perl`.

Relative Progress Graphs

To better explain and compare the errors seen in Figure IV.8, we now examine the relative progress of each through execution using a *Relative Progress Graph*.

Figures IV.11 and IV.12 compare the baseline execution of program pairs with the estimated execution derived from sampling. A point plotted at (x, y) indicates that when thread 0 has executed x instructions, thread 1 has executed y instructions. The solid line represents the baseline detailed simulation and the broken lines represent the progress estimated by our various sampling methods.

The slope of the line at a point indicates the relative IPCs of the two programs, but actual IPC values cannot be derived from this graph because it does not show how many cycles have executed. The graph can be used to verify that the sampled runs enter the correct co-phases together. The ideal behavior is for the sampling approach to identically follow the baseline run.

In Figure IV.11 we examine the co-execution of `bzip2` and `gcc`. This pairing has the most dramatic variation in relative IPCs amongst our test cases, as can be seen by the nearly horizontal and vertical line segments midway through execution. Each sharp bend in the graph represents a transition between significantly different co-phases. It is immediately obvious that the last sample method does not track the behavior of the baseline (the top line in the graph). It does a reasonable job of tracking the sharp bend at 2.5 billion instructions, but because it has already deviated from the baseline it does not match the nearly horizontal phase for even a third of its length, an error which cannot be corrected. The static method (the bottom line in the graph) sampled unfortunately just before the nearly vertical section, causing it to switch to the horizontal too late and hence continue in that co-phase too long. Despite this error it manages to track subsequent phase changes fairly accurately. All of the other methods make better use of phase knowledge, ensuring that they take samples at the phase combinations. The slight deviation from the baseline is determined by how representative of the entire phase are the samples. The close tracking of the baseline indicates that sufficiently many phases were used for both programs.

Figure IV.12 shows a poorly-performing pairing with about 10% error in the non-static methods. Here the phase behavior settles into an undulating line after the first billion instructions. Although Last 5% sampling looks the best here, it can be seen that this is almost an accident. It crosses the baseline many times but rarely has a similar slope; the frequent variations caused by `bzip2`

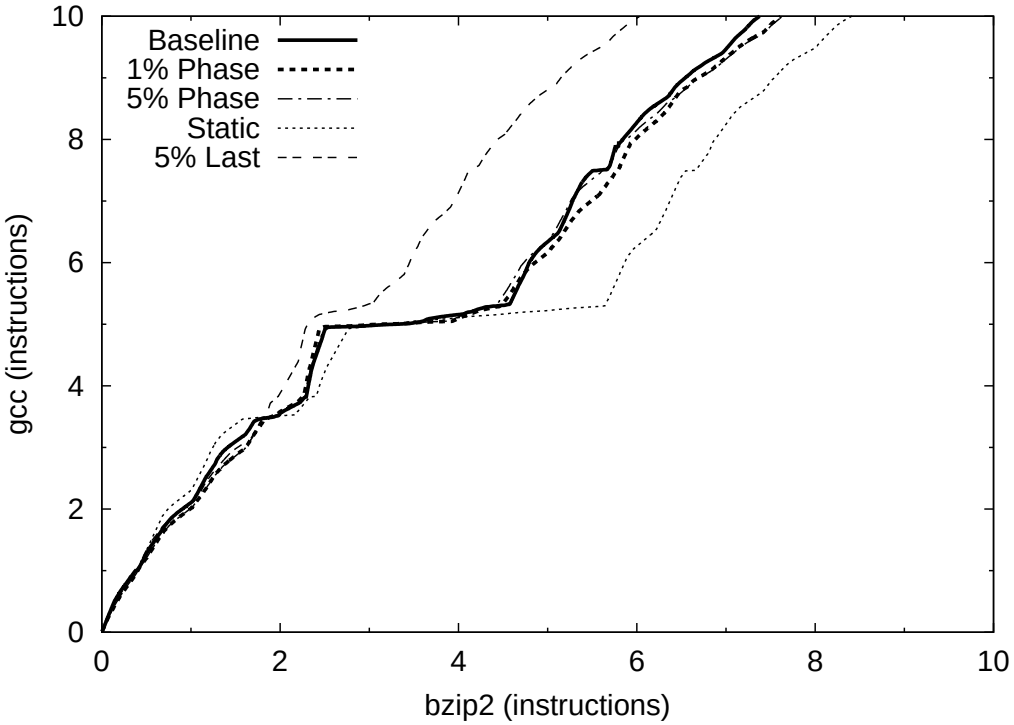

Figure IV.11: Relative progress for `bzip2-gcc`.

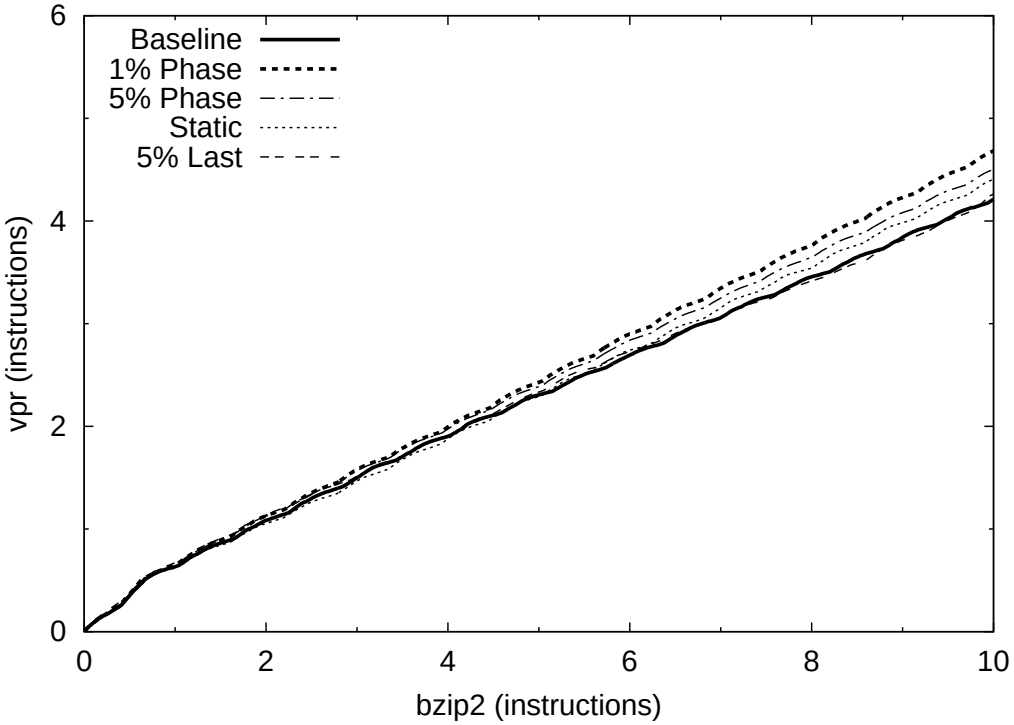

Figure IV.12: Relative progress for `bzip2-vpr`.

repeatedly send it back towards the baseline but it does not correct its course again until it is too late. These errors make its IPC estimate no better than that of the other techniques. The phase-based techniques track changes in the baseline admirably, but these runs consistently diverge over time, suggesting that the samples are slightly off from the average trend. The static method gets an advantage in this case by not taking the earliest samples, keeping its error in `vpr`'s execution much lower.

In each case it is clear that attention to phases provides essential information for tracking multithreaded execution behavior. Regular sampling without phase information succumbs to error for reasonable sampling frequencies. Shorter samples would allow more frequent sampling but would decrease sample accuracy; this will only give reasonable results for program pairs with consistent behavior such as `bzip2-vpr`, but not for complex combinations such as `bzip2-gcc`.

The analysis for `bzip2-vpr` indicates that advanced warmup techniques may improve our results. Functional warming during fast-forwarding, as used by Wunderlich *et al.* [65] is a promising technique. Keeping caches and branch predictors warm has a modest impact on the performance of fast-forwarding, so the increased execution time should not outweigh the anticipated benefits to accuracy. Multithreaded functional warming requires that state updates from the threads be interleaved, so each thread will need to be fast-forwarded in small increments according to their estimated relative IPCs.

Variability of Co-phase Samples

We now examine the benefit of using co-phase information to guide sampling. We find that samples taken from co-phases exhibit less variation than samples taken across all co-phases during our baseline runs. We break the execution of an SMT workload into 10M combined instruction intervals. We then

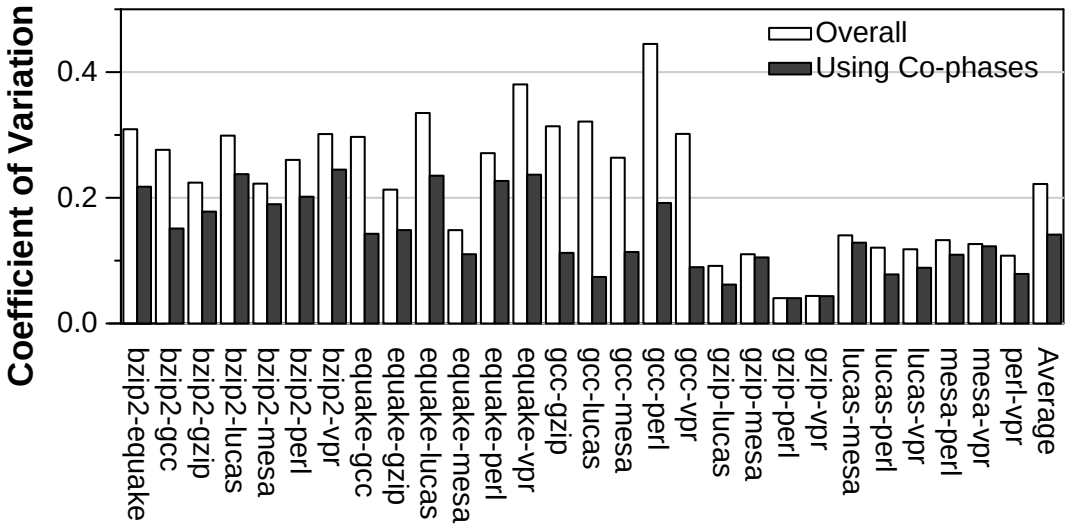

Figure IV.13: Coefficient of variation improvements through phase separation.

calculate the Coefficient of Variation (CoV) over all of these intervals of execution, as shown in Figure IV.13. This represents the variation seen when randomly sampling over the complete baseline execution of the workload.

Next, we compare this to the variance in samples seen when guiding sampling with the co-phase matrix. To calculate this, we bin all of the SMT workload intervals into co-phase matrix entries, and we calculate the CoV of each co-phase matrix entry. We then weight each CoV based upon the amount of execution each phase accounted for. These weighted CoVs are then combined to arrive at an average CoV that would be seen when gathering samples based upon the co-phase matrix. The co-phase CoV is also shown in Figure IV.13. The results show that the co-phase sampling always made an improvement, reducing the variance by one-third on average.

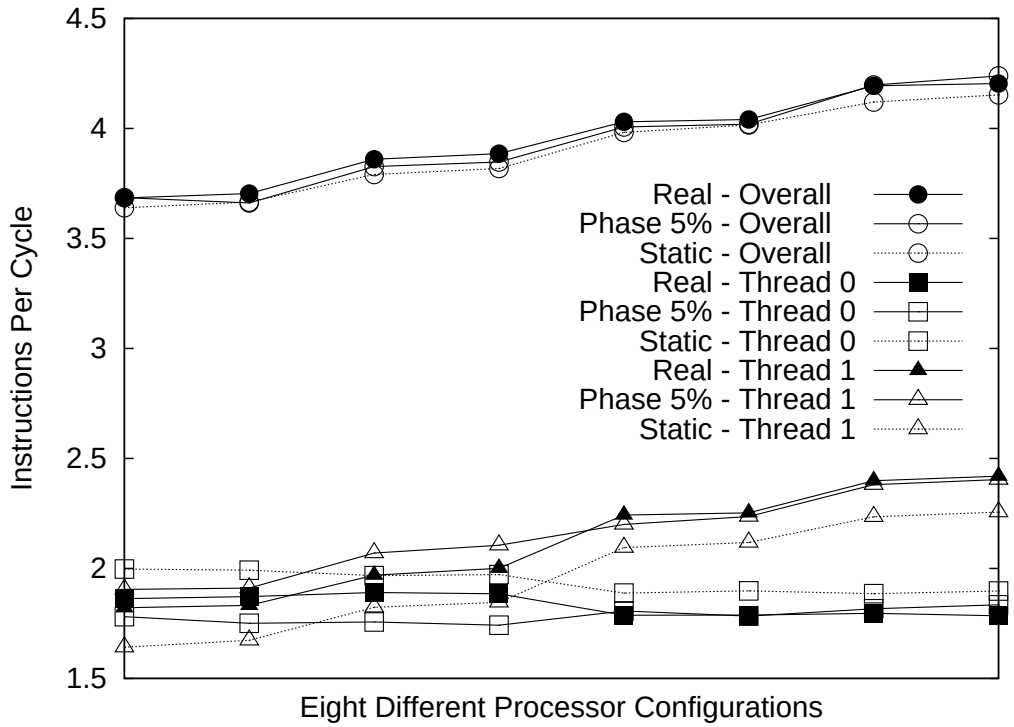

Figure IV.14: Overall and per-thread performance for `bzip2-gcc` under different architecture configurations.

IV.B.9 Relative Error

We also verified that the co-phase methods are suitable for comparing different simulator configurations. We ran different pairs of programs through eight configurations, varying the size of the L1 caches, the L2 cache and the branch predictor tables. Timing parameters were changed so that the smaller structures had lower latency. The `bzip2-gcc` pair was particularly sensitive to architectural changes so in Figure IV.14 we show real and estimated overall and per-thread IPC. The Static Phase method is particularly consistent in its error, and the magnitude of the error is relatively constant over most of the configurations. It changes at the 5^{th} processor configuration, where the error drops to nearly zero.

Table IV.3: IPC and number of co-phases found for each set of four programs.

Programs	Instructions Per Cycle					Co-phases	
		Per Thread					
	Overall	0	1	2	3	Possible	Observed
bzip2-equake-gcc-lucas	5.1	1.4	1.1	1.7	0.9	15552	469
bzip2-gcc-mesa-vpr	4.6	1.2	1.6	1.2	0.5	2688	305
equake-gzip-lucas-perl	4.0	1.0	1.3	1.0	0.8	1944	375
gzip-mesa-perl-vpr	3.6	1.2	1.4	0.6	0.5	336	62

IV.B.10 Four-Context Simulation Results

We also examined applying our co-phase matrix approach to a four-context SMT processor. Table IV.3 shows the the overall IPC and per-thread IPC for the four program combinations we examined. The last two columns show the total possible number of co-phase combinations and the number observed during simulation.

Figure IV.15 shows the overall IPC error rates for the four program combinations we examined. We compare the results for Single, 1% Phase, 5% Phase and Static methods. The results show that when using all four contexts the machine is completely resource constrained so errors in different threads balance out, making the overall IPC error negligible.

Figure IV.16 shows the detailed breakdown of per-thread error for one combination. Results are shown across the different sampling techniques for the overall error, and per-thread error for `equake`, `gzip`, `lucas`, and `perl`. As in the two-threaded case, `perl` is a significant source of error because of the limited number of phases found. Static errors are more pronounced than the other methods, unlike the situation with two threads, because the larger number of programs produce shorter co-phases, increasing the probability that a the static sample will not match the dynamic instances of the co-phase. This is similar to

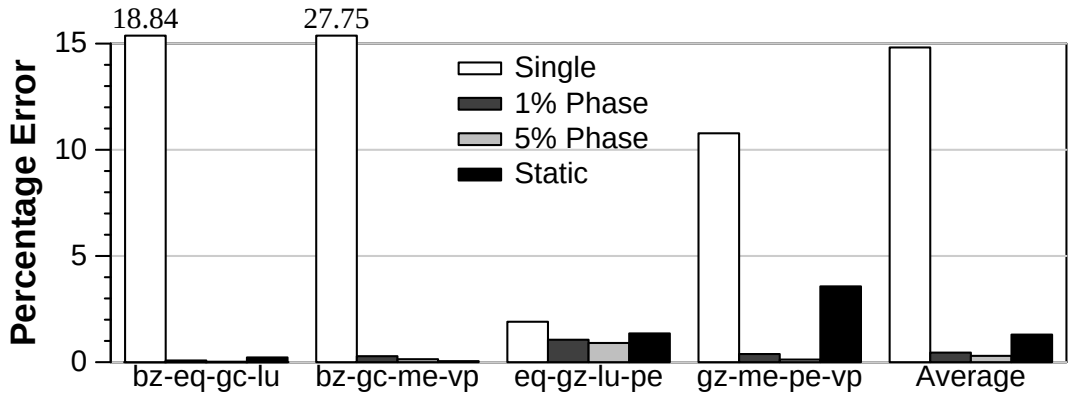

Figure IV.15: Overall IPC error rates for four four-threaded combinations.

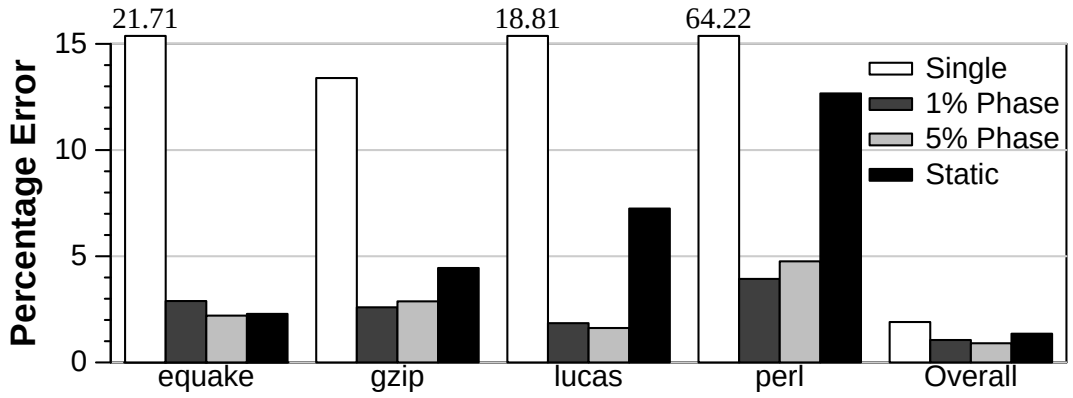

Figure IV.16: Per-thread IPC accuracy for the `equake-gzip-lucas-perl` combination.

the problem noted with `gcc`'s frequent phase changes, but worse. Nonetheless, the magnitude of the errors is smaller than in many two-threaded cases because more of the simulated machine's resources are used at capacity, allowing less variation in performance.

IV.B.11 Revised Methodology For the Static Co-Phase Method

After our initial work on the co-phase method we were able to improve both our technique and analysis for the static co-phase method. For the remainder of this chapter we present results that take additional simulation resources

(allowing us to create more baseline measurements), our experience with reduced checkpoints, an improved version of SimPoint and refinements on our sampling technique.

We analyzed the benchmarks using SimPoint 3.0 [19]. The phase analysis for our results covered all of program execution and used intervals of 5M instructions. We limited the maximum number of phases found by SimPoint to 30, *i.e.*, Max K set to 30. The fewest number of phases found was `mesa` with 25 phases. So, all of the benchmarks used between 25 to 30 phases.

For increased accuracy and decreased simulation time, we improved our handling of warmup. We use the hit-on-cold warmup technique along with the following strategy to further reduce the impact of cold-start effects. We ignore the first 1.5M instructions of execution and use the remaining instructions to calculate the contents of the Co-Phase Matrix. Data collection ends once one thread commits 3.5M instructions beyond the detailed warmup. This ensures that we don't leave the simulation point interval in either thread and discards simulation data that is tainted by warmup problems.

Figure IV.17 shows the number of co-phases in the static co-phase matrix for the various benchmark pairs. Populating the static co-phase matrix with performance numbers requires computing performance numbers for all of these co-phases. The number of co-phases varies from 650 to 870, with 759 co-phases on average. To efficiently compute these co-phases, we employ the reduced checkpointing techniques as discussed in Chapter III. These reduced checkpointing techniques minimize the architectural state that needs to be stored on disk per simulation point. The shorter simulation period (at most a quarter of the number of instructions used previously) and smaller checkpoints kept the detailed simulation time manageable.

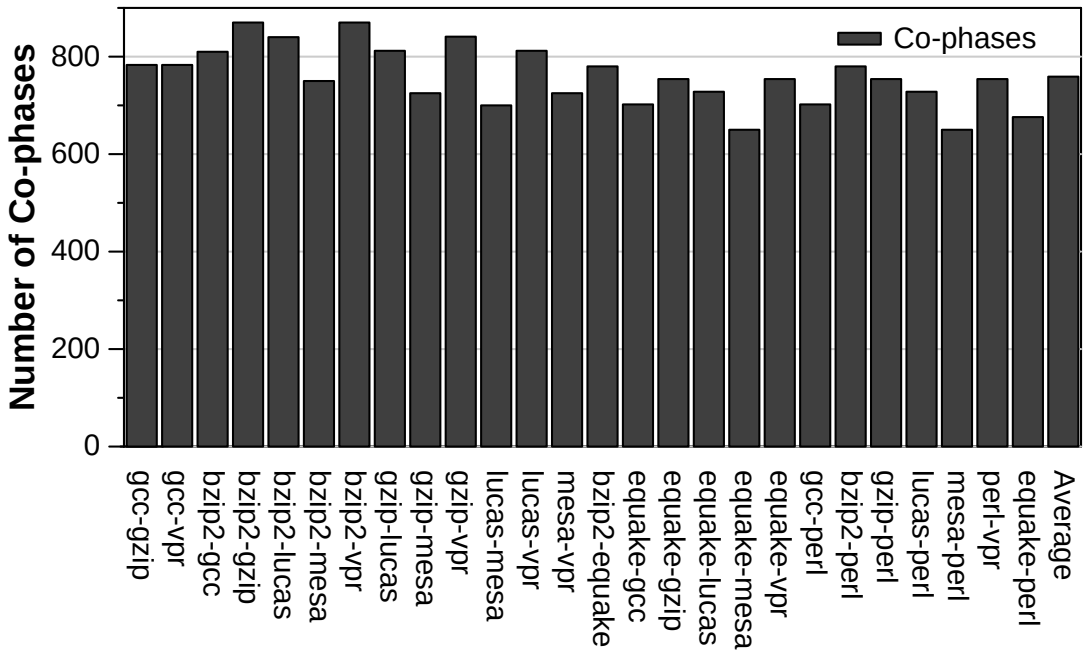

Figure IV.17: Number of co-phases per benchmark pair.

IV.B.12 Single Starting Point Co-Phase Matrix-Driven Simulation

Our first evaluation step was to verify that the new methodology produced comparable error rates for 2 and 10 billion instructions. Using smaller evaluation intervals allowed us to evaluate the static co-phase method more comprehensively than in our earlier experiments.

Figure IV.18 validates the improved static co-phase matrix approach with respect to the full detailed simulation runs for 100 starting offset pairs. For this result we simulated each starting offset for 1 billion detailed instructions to get the baseline CPI starting at random offsets. We then used the static co-phase matrix method to determine the estimated CPI until both programs reached 1 billion instructions of estimated execution. This gave us an estimated error for each pair of starting points. We report the average error across all starting points in Figure IV.18 for the set of programs as before. For this result, 10 starting offsets

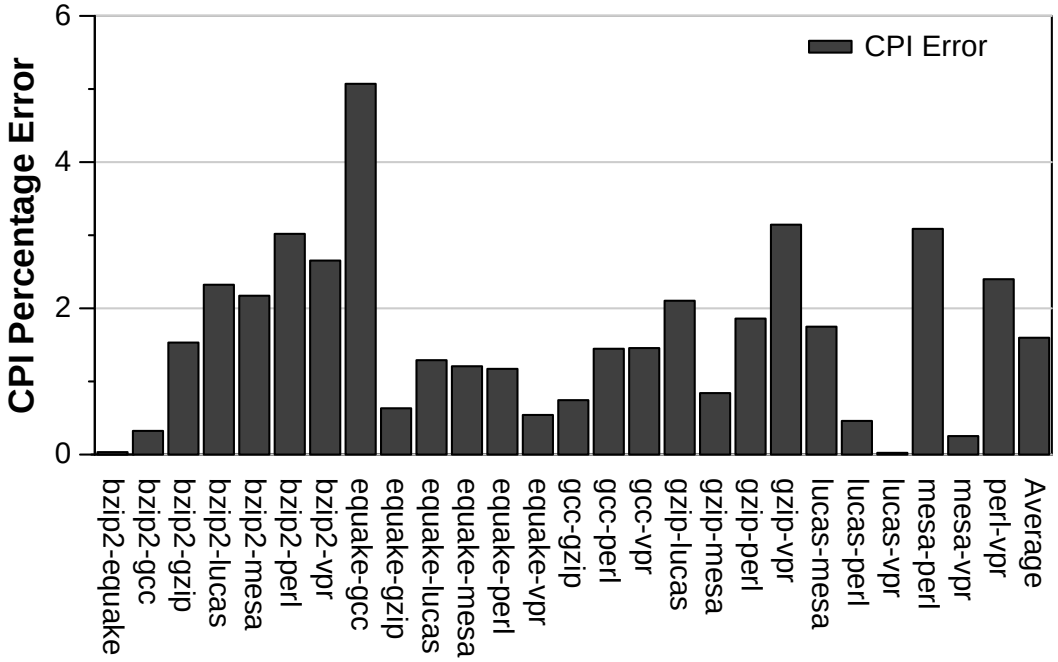

Figure IV.18: Error in CPI for static co-phase method simulation.

were chosen randomly for each program, which resulted in 100 simulations for a benchmark pair.

The results in Figure IV.18 show the average performance prediction error over those 100 simulation runs. We observe an average error of 1.6% and a maximum error of 5.1%. The highest error is for `equake-gcc`; others have error of at most 3.1%. The benchmark pair `equake-gcc` presents challenges due to dramatically different execution rates of the two programs at a particular point, which can magnify small errors in sampling. We investigate this in more detail in Chapter V. Additional experiments showed that this combination of programs benefits from a longer sampling period. Our low error rates validate our changes to interval size, sampling and phase analysis. They also demonstrate that the co-phase matrix can be used to accurately estimate performance when running a combination of programs on a multithreaded processor from a single starting point.

IV.C Summary

Simultaneous Multithreading architectures are appearing in commercial processors, yet there is still relatively little support for sampling or determining where to simulate to achieve representative simulation results. The challenge in creating a sampling approach for multithreaded simulation is in determining how far to fast-forward each individual thread between samples. This distance will vary between different architecture configurations and as the threads execute through different phases of execution.

In this chapter, we presented the co-phase matrix method for sampling the execution of Simultaneous Multithreading machines. Our simulation approach builds a co-phase matrix and uses it to guide fast-forwarding between samples. In performing detailed simulation using the co-phase matrix, we were able to estimate the IPC for multi-program workloads with an average error of 4% when using the 1% co-phase sampling approach. Our static co-phase method allows parallel simulation and can estimate workload performance from all possible thread starting positions with just 4.3% error. Subsequently, we improved this error rate to 1.6% by using a more precise phase analysis that increased the number of co-phases but allowed us to simulate them for shorter periods of time.

IV.D Acknowledgement

This chapter contains material from *A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation* [59], in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Michael Van Biesbrouck, Timothy Sherwood and Brad Calder. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from *Considering All Starting Points for a Simultaneous Multithreading Simulation Methodology* [56], in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are

V

Benchmark Suite Performance

The co-phase matrix approach in Chapter IV focused on providing an accurate simulation assuming a single starting position in each binary being simulated. In this chapter we show that the architecture behavior and overall throughput can vary drastically based upon the different starting points within the different benchmarks. Therefore, to completely evaluate the effect of a multithreaded architecture optimization on a workload, we may need to simulate many or all of the program combinations from different starting offsets. Exhaustively running all program combinations from many starting offsets is infeasible — even running single programs to completion is infeasible with modern benchmarks.

This chapter proposes an efficient multithreaded simulation methodology that estimates average performance over all starting points when running multiple programs on a multithreaded processor. This is achieved by populating the co-phase matrix with performance results for the different co-phases as was done in Chapter IV. Once this co-phase matrix is populated, we use it to estimate the average performance over all starting points. This is done by randomly picking a number of starting points and by analytically simulating each of these co-phase executions with their given starting points. Since the analytical simulation is done very efficiently, the whole multithreaded simulation for a

set of starting points completes very quickly, in at most a few minutes. To the best of our knowledge, this work is the first to propose a multithreaded simulation methodology that estimates average performance for all starting simulation points. The most similar approaches require real hardware to execute entire benchmarks repeatedly until performance statics converge [53, 60, 61].

We believe that this is the best way to find the average performance of a pair of programs and propose a weighted average of the statistics for all pairs of benchmarks in a benchmark suite as an overall metric of multithreaded processor performance. The drawback to this technique is that constructing co-phase matrices for every pair of programs is time-consuming, and the number of instructions to be simulated for a workload can be more than a researcher has time to analyze when doing design-space exploration. Analyzing all pairs of reference inputs for the whole SPEC CPU2000 suite would require about 3.5 trillion instructions to be simulated.

Due to the high cost of our previous metric, we propose an approximation to the metric that still incorporates co-phase performance and the relative weight of phases in the benchmarks. First, we determine the most significant co-phase behaviors in a benchmark suite. To make this possible, we propose a technique based on *Principal Components Analysis* (PCA) and *Cluster Analysis* for reducing the required number of *co-simulation points*. The proposed technique detects similarities in phases from different benchmarks and different inputs to the same benchmark. Additional savings come from the elimination of rare behaviors that do not contribute significantly to performance estimates and by avoiding fine-grain simulation of co-phase behavior. As we also introduce the use of co-phase interpolation, the number of combinations to simulate can be kept small. It is possible to use just 50 co-simulation points for two-thread workloads for SPEC CPU2000 (2.5 billion instructions in total) to accurately estimate the

throughput of all benchmark combinations.

Our approach uses long enough samples to avoid most warmup problems. In addition, we insure that the samples contain homogeneous behavior relative to the sample size. Homogeneous behavior within a sample allows for faithful comparisons of simulation results across processor architectures. The reason is that when comparing performance numbers across multithreaded processor architectures, the various threads may have different progress rates. Homogeneous behavior allows for stopping the simulation at any point while yielding representative co-thread performance numbers.

Our final technique is scalable to architectures with many cores and contexts per core. PCA and improvements to the clustering methodology allow us to use many threads when identifying thread combinations to simulate, previously a scaling limitation. In the future, this should allow us to handle large benchmark suites with more than two threads running at a time while using a limited amount of simulation time.

V.A Background

This chapter has two contrasting goals, accurate representation of the diverse of behaviors that exist even within a single benchmark pair and exploiting similar behaviors within pairs of benchmarks to reduce simulation time. In preparation for this, we examine how differently to programs can behave when running together if we vary their relative start times and the section of execution to be simulated. Then we consider how PCA is used to find overlaps within benchmark suites.

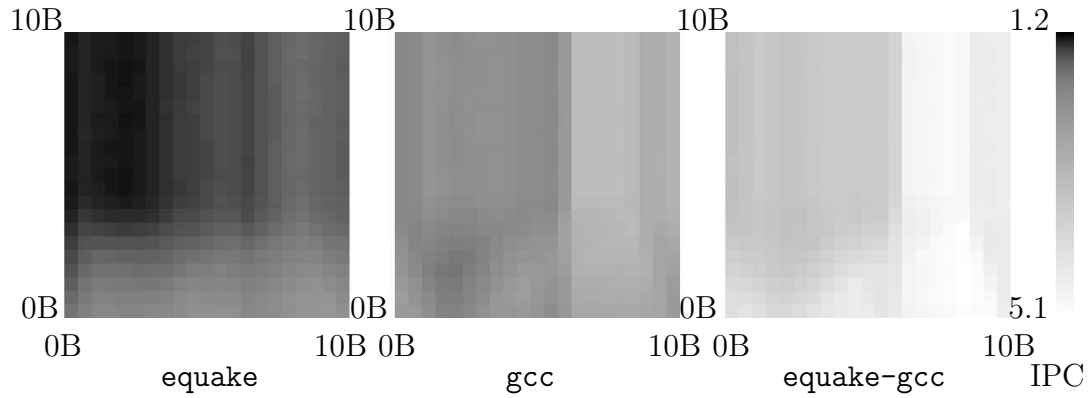

Figure V.1: The graphs show the IPC when `equake` and `gcc` are run together from various starting offsets. There are graphs for each program's IPC and their combined IPC. The shade of gray at (x, y) indicates IPC when simulation starts with `gcc` x instructions from the start of its execution and `equake` y instructions from the start of its execution. Simulation completed after a total of 10 billion instructions were committed.

V.A.1 Starting Offset Effects in SMT Simulation

Kihm *et al.* [26] showed that SMT performance is sensitive to starting points. They profiled a number of co-program executions with different starting points and observed that different performance results were obtained. Their study was done on real hardware, namely on an Intel Pentium 4 processor. Unlike this dissertation, however, they did not provided a simulation methodology that allows for capturing the average performance for all starting points. Here we use detailed simulation and the static co-phase method to demonstrate the effects of simulation starting points.

Most studies use absolute performance estimates based upon a small number of simulation runs to predict the effects of microarchitectural changes. These simulation results are not completely accurate in even the single-threaded case, so some simulation methodologies focus only on the change in performance metrics due to microarchitectural changes, not the absolute numbers produced.

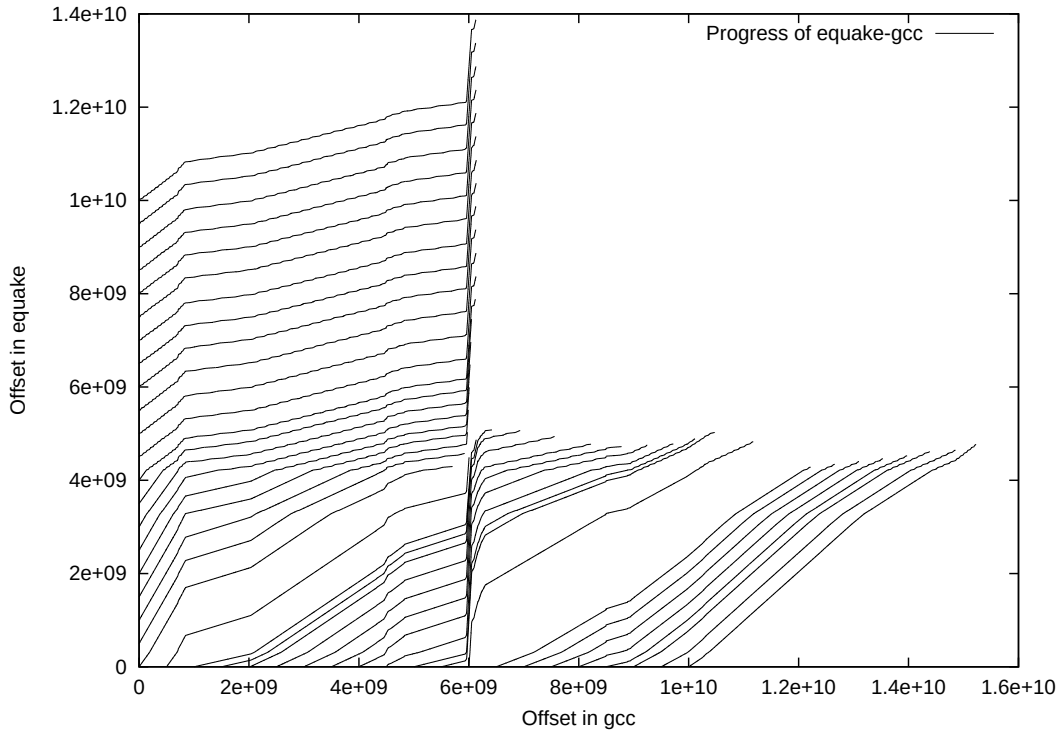

Figure V.2: Relative progress of `equake` and `gcc`. Each line represents a single 10B-instruction execution of `equake-gcc` from a different starting offset (either equake or gcc is always run from the beginning). Each plotted point represents execution offsets that occur during SMT execution.

We show that both approaches can lead to misleading results when simulating SMT processors.

Absolute Performance Predictions

A single starting point per thread might not be representative for what one would observe in a real multithreaded environment. The pitfall of selecting a single starting point is that the performance results that you obtain may be very different from those with different starting points. This is illustrated in Figure V.1. This plot shows the average IPC that is obtained when simulating two benchmarks, `equake` and `gcc-166`, from different starting locations. In each of

these experiments, we simulate until the threads execute a total of 10B instructions. We show results for 441 different relative offsets; adjacent sample points differ by 500M instructions in one thread's starting offset. The average (aggregate) IPC numbers are encoded by shades of gray: white means an IPC of 5.1 whereas black means an IPC of 1.2; we provide a scale to estimate intermediate values. We clearly observe that the overall performance is very sensitive to the starting points; the overall IPC varies from 1.2 to 5.1. So, the pitfall is that using a single starting point may impact expected performance results significantly.

To examine the behavior of multiple starting points further, we now examine the relative progress of execution of several different starting points using a relative progress graph. Figure V.2 shows the relative progress graph for `equake` and `gcc`. The relative progress for `gcc` and `equake` are shown along the horizontal axis and vertical axis, respectively. For the line that starts at $(0, 0)$, a point plotted at (x, y) indicates that when thread 0 has executed x instructions, thread 1 has executed y instructions. The other lines start at different points on the graph, since they represent either `gcc` starting at the beginning of execution, and `equake` starting simulation at one of the offsets shown on the vertical axis. Similarly, the lines starting on the horizontal axis represent `equake` starting simulation at the beginning and `gcc` a given number of instructions (shown on the horizontal axis) into `gcc`'s execution.

In Figure V.2 we see that adjacent starting offsets usually produce similar but not identical executions. The executions that start near $(0, 0)$ are the ones with the most variety until `gcc` reaches the 6B-instruction mark. At that point `gcc` makes much less relative progress than `equake`. The reason is that `gcc` suffers from a large number of L2 misses at that point. The endings of many executions are thus dominated by progress in `equake`. This extreme phase behavior also appears very clearly in the `gcc` graph in Figure V.1 as a sharp increase in

IPC for simulations that start `gcc` at an offset of more than 6B instructions. It is less obvious in the `equake` graph because `gcc` is much more affected than `equake`.

The sharp change in `gcc`'s performance is worth additional investigation as it improves understanding of the interactions between programs on SMT processors. In Figure V.3 we plot the performance of the programs, measured in instructions per cycle (IPC). The solid lines show SMT behavior and the broken lines show single-threaded behavior. The `equake` lines are marked with squares and the `gcc` ones with triangles. The x-axis indicates the proportion of instructions committed; in the case of the SMT executions this is the combined number of committed instructions between the two threads, so the SMT executions are aligned with each other but the single-threaded and SMT executions of a particular thread are not. The markers are placed every billion instructions of single-threaded execution to help the reader match up parts of execution between singled-threaded and SMT runs. The interesting event that occurs 6B instructions into `gcc`'s execution is at the 50% mark on both graphs; this point is easily identified by dramatic changes in `gcc`. We can see that single-threaded execution also reaches a low IPC at this point, but for a much briefer period. The cause is visible in Figure V.4, which uses a log scale to show the average number of L2 cache misses per single-threaded instruction committed. Here, the single-threaded `gcc` briefly misses one cache access per 20 instructions, but the SMT execution misses one of every four instructions due to the increased cache contention. The extremely high miss rate allows `equake` to continue on while `gcc` is nearly stalled. Note that while `equake` often has higher miss rates in SMT execution than when single-threaded, the absolute frequency of misses is much lower so there is less of an impact.

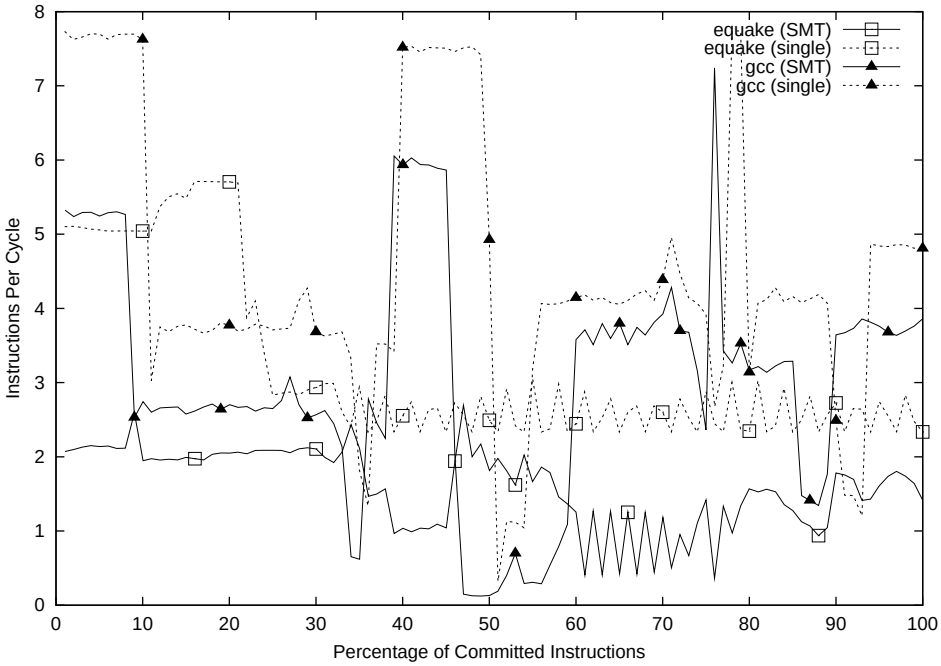

Figure V.3: IPC of `equake` and `gcc` running singly and as a pair.

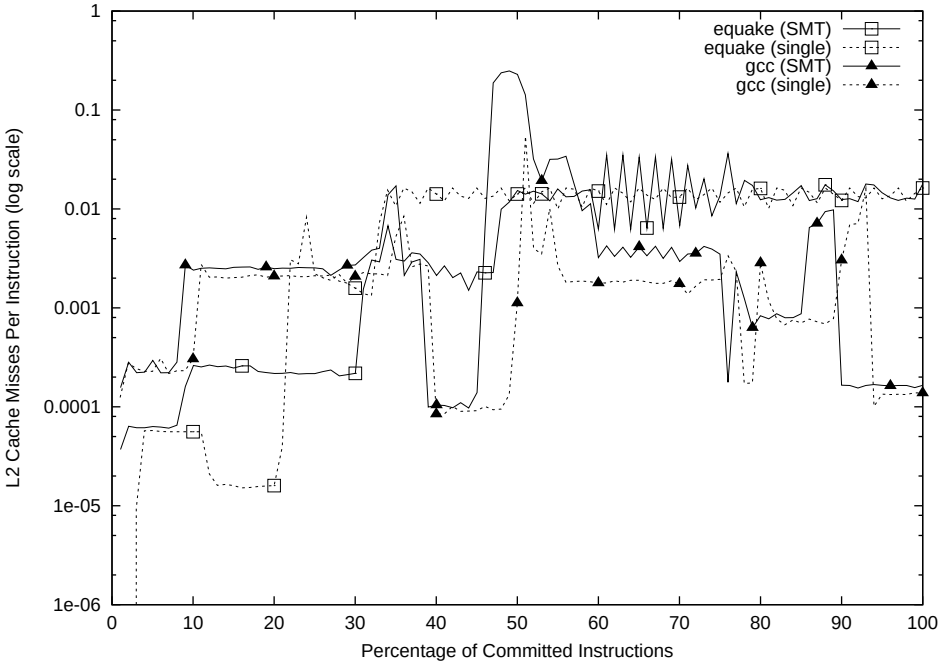

Figure V.4: L2 cache miss behavior of `equake` and `gcc` running singly and as a pair.

Relative Performance Predictions

In many practical circumstances, being able to accurately estimate relative performance change is more important that absolute performance estimation [15, 39, 59]. In other words, absolute simulator and methodology accuracy is less important than relative accuracy as long as the relative effects of hardware changes are faithfully tracked. This is especially the case for early design stage studies.

In the case of multithreaded simulation, this means that the variation in performance for different simulation starting points is not important as long as all of the starting points are equally affected by microarchitectural changes. To examine this, we simulated 20 pairs of starting offsets each for three pairs of programs and executed each of these on eight different hardware configurations, for a total of 480 experiments consisting of 1 billion committed instructions for each run. The program pairs and machine configurations are the same as those used to evaluate relative error in Chapter IV. The hardware configurations cover all combinations of small and large L1 caches, L2 caches and branch predictors.

We expect that for any hardware change, some benchmark pairs will see performance gains whereas others will not. If we take a single starting point for a particular benchmark pair, then performance will clearly get better or worse as the hardware configuration changes. Unfortunately, picking a different starting offset might give us the opposite result for the same hardware configurations. To examine this we take all 28 pairs of different hardware configurations, and we see how the performance for each pair of hardware configurations differ for a two-program workload examining 20 different starting offsets for that workload. For a pair of hardware configurations, we take a pair of programs and we vary the starting offsets for those programs 20 times. The result of each run is a ranking of the two hardware configurations saying that one has more throughput

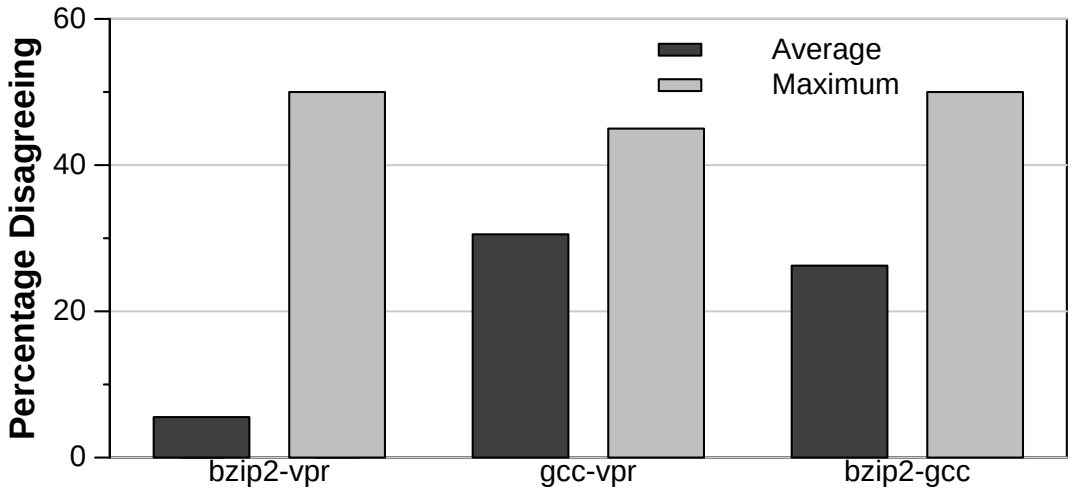

Figure V.5: Performance effect disagreement after hardware configuration change. 0% indicates that all starting offsets improve (or degrade) due the change; 50% indicates that half improve and half degrade, the worst possible result.

than the other. We then calculate for the 20 starting offsets examined with two hardware configurations, what percentage of time did the ordering of the two architectures agree versus disagree. If they were always identical, then we would see 0% disagreement. The worst case result would be 50% disagreement, which means that 50% of the time the first hardware configuration was said to be better than the second, and *vice-versa* and the only cause of this would be using different starting offsets.

Ideally, we want to see 0% disagreement as a result of the experiment. With 8 hardware configurations, there are 28 distinct pairs of configurations that we compare. Figure V.5 shows the percentage of offsets that disagree in the direction of improvement with the majority of experiments. Since this number varies over the 28 possible hardware configuration pairings (potential experiments), we report average and maximum disagreement rates. The possible rates range from 0% (best) to 50% (worst).

We find that `bzip2-vpr` consistently favors one hardware configuration over the other in all but a couple possible experiments, whereas the others typi-

cally have results divided between the two hardware options. In the architecture comparison that caused the most disagreement, the average performance change was over 2% and frequently much higher. For each pair of programs, there are experiments that would produce misleading results if only a single starting offset were chosen. For most experiments, we can expect that some starting offsets lead to improvements but others do not. We conclude that researchers examining relative performance effects still need to be able to analyze the distribution of the results over many starting offsets rather than starting execution from a single offset.

V.A.2 Evaluating Benchmark Suites with PCA

Our work on using PCA with multithreaded workloads uses novel techniques but it is based on past single- and multithreaded PCA workload reduction research.

Single-Threaded Workload Analysis Through PCA

Eeckhout *et al.* [14] proposed a workload reduction approach that picks a number of program-input representatives from a large set of program-input pairs. They first measure a number of program characteristics of the complete execution for each program-input pair. They subsequently apply principal components analysis (PCA) in order to get rid of the correlation in the data set. (In section V.B.7 we will discuss why this is important.) As a final step, cluster analysis (CA) computes the similarities between the various program-input pairs in the rescaled PCA space. Program-input pairs that are close to each other in the rescaled PCA space exhibit similar behavior; program-input pairs that are further away from each other are dissimilar. As such, these similarity metrics can be used for selecting a reduced workload. For example, there is little benefit in

selecting two program-input pairs for inclusion in the reduced workload if both exhibit similar behavior.

This initial work on workload reduction used microarchitecture-dependent and microarchitecture-independent characteristics as input to the workload analysis. The main disadvantage of using microarchitecture-dependent characteristics is that it is unclear whether the results are directly applicable for other microarchitectural configurations. Phansalkar *et al.* [41] made a step forward by choosing microarchitecture-independent characteristics only. This makes the reduced workload more robust across different microarchitectures. Eeckhout *et al.* [13] further extended this workload analysis approach by looking into similarities between program-input pairs at the phase level. Instead of measuring aggregate microarchitecture-independent metrics over the complete benchmark execution, they measure those metrics at the phase level. They subsequently used the PCA/CA workload analysis methodology to identify similarities across the benchmarks and inputs at the phase level. The end result from their analysis is a set of representative phases across the various program-input pairs; these phases along with an appropriate weighting, allow them to make accurate performance estimates of the complete benchmark suite.

All of this prior work focused on finding representative workloads for single-threaded processor simulation. This chapter uses and extends the PCA/CA workload analysis methodology based on microarchitecture-independent metrics to select representative co-phases to be simulated in an accurate multiprogrammed multithreaded processor simulation methodology.

Multithreaded Simulation Methodologies

Raasch and Reinhardt [42] used an improved SMT simulation methodology in their study on how partitioned resources affect SMT performance. They

selected a set of diverse co-sample behaviors rather than randomly chosen co-sample behaviors. First, they find single simulation points using SimPoint [48]. They then run all possible two-context co-phase combinations on a given microprocessor configuration — in their setup they ran 351 co-phases. For each of those co-phases, they compute a number of microarchitecture-dependent characteristics such as per-thread IPC, ROB occupancy, issue rate, L1 miss rate, L2 miss rate, functional unit occupancy, *etc.* Using the methodology from [14], they then apply principal components analysis (PCA) and cluster analysis (CA) to come to a limited number of 15 two-context co-phases. There are at least three pitfalls with this methodology. First, a single simulation point is chosen per benchmark. This could give a distorted view for what is being seen in a real system where programs go through multiple phases. Second, the single simulation points selected by SimPoint may represent heterogeneous program behavior which makes comparing co-phase behavior across processor architectures questionable — different portions of the workload may be executed under different processor architectures. To address this issue, we consider multiple simulation points with homogeneous phases. Third, this approach is driven by microarchitecture-dependent characteristics. As a result, the distinct co-phase behaviors obtained through PCA and CA will be representative for the processor architecture for which the characteristics were measured. However, it is questionable whether these co-phases will be representative when applied to other processor architectures. In this chapter, we address this pitfall by considering microarchitecture-independent characteristics for determining representative co-phases.

V.B Discussion

First we develop a method to analyze benchmark pair performance from all possible simulation starting points and then we reduce the complexity of

examining all pairs of benchmarks in a benchmark suite using PCA and improved clustering techniques.

V.B.1 All Combination Performance

As extensively discussed in Section V.A.1, using a single starting point in multithreaded simulation could be misleading. This observation argues for an approach in which all starting points are considered for estimating overall multithreaded performance. In this section, we now describe how this can be done using the co-phase matrix described in the previous section.

To generate a performance estimate of all combinations of starting offsets, we use the same method for a single pair of starting points as we described in the previous section, but run it for many starting points. This creates a metric, called the *All Combination* (AC) performance number, which represents the average performance of a pair of benchmarks, independent of particular starting offsets. In this dissertation we only examine the AC in terms of overall CPI, but any other processor statistic can be collected in the same way. The AC is the average performance found when executing both programs for one billion instructions from every combination of possible program offsets. If a program reaches the end of its execution, it is restarted at the beginning to avoid bias near the start and the end of programs. Fixing an execution length is necessary to make averages meaningful and easy to compute. Choosing a long execution length ensures that the weighting of co-phases will match that of continuous multithreaded execution.

Although the static co-phase method for a single combination of starting addresses is fast, running it for all possible starting points obviously is infeasible. In our setup, this would require 10^{23} analytical simulations using the static co-phase matrix per pair of SPEC benchmarks. Although an analytical simulation

using the static co-phase matrix is extremely fast, simulating that many runs would be impossible to do. Thus we propose to sample the set of possible starting offsets. We examine two sampling strategies, namely random and stratified sampling. For both approaches, we assume a populated static co-phase matrix to start from as described in Section IV.B.11.

We use random sampling to pick starting points for both threads. For each pair of randomly selected starting points, the static co-phase method is used to estimate performance when executing both threads from the given starting point. By doing this for a sufficiently large number of randomly selected starting points, called *samples*, an average performance estimate can be computed for *all* possible starting points. We always simulate for 1B instructions of combined execution, so the average CPI over all samples is just the simple average of all collected CPI rates. (Other metrics may require more complicated computations.) An interesting property of random sampling is that we can estimate the variability of the samples, which allows us to provide confidence bounds for average performance estimates. In addition to pure random sampling, we also consider stratified random sampling to ensure even coverage of possible starting points.

V.B.2 Convergence of All Combination Performance Estimates

In this section we evaluate our newly proposed multithreaded simulation methodology. Using the improved static co-phase methodology from Section IV.B.12, we show that the co-phase matrix can also be used to estimate multithreaded performance for all starting points.

We now examine the all combination performance for 26 benchmark pairs using the static co-phase method and two sampling techniques, random sampling (Figure V.6) and stratified random sampling (Figure V.7). For this

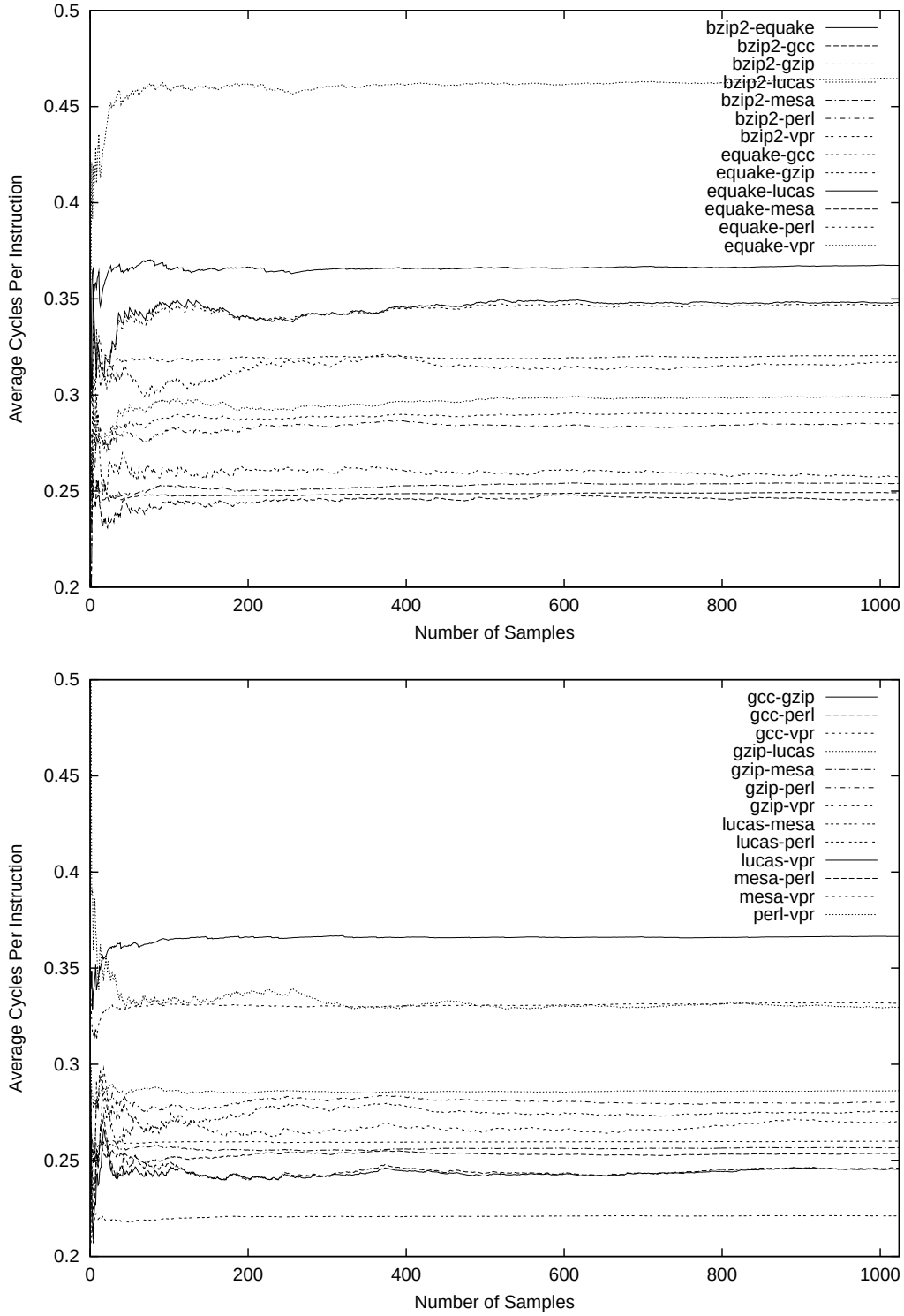

Figure V.6: All combination CPI convergence using random sampling.

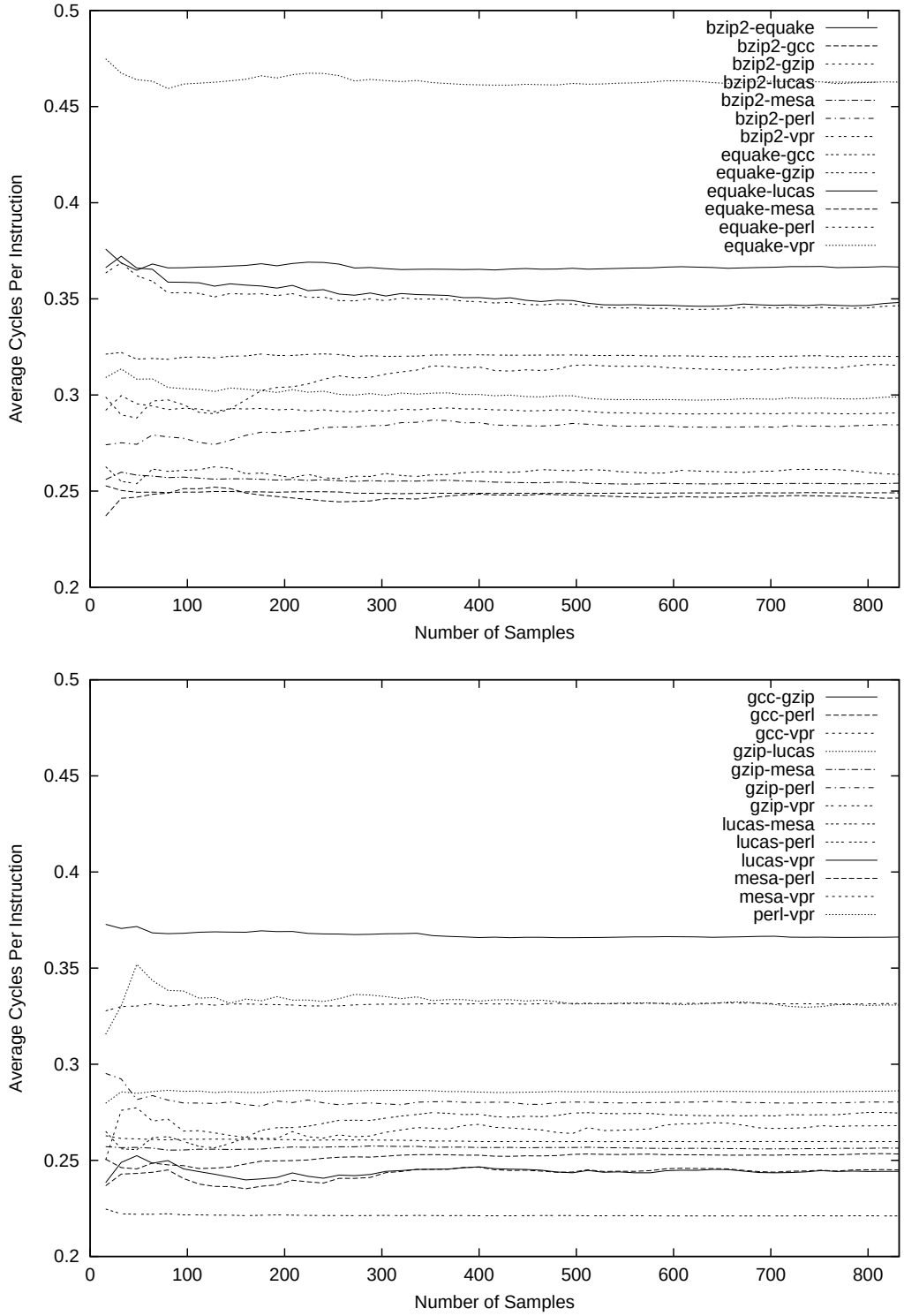

Figure V.7: All combination CPI convergence using stratified random sampling.

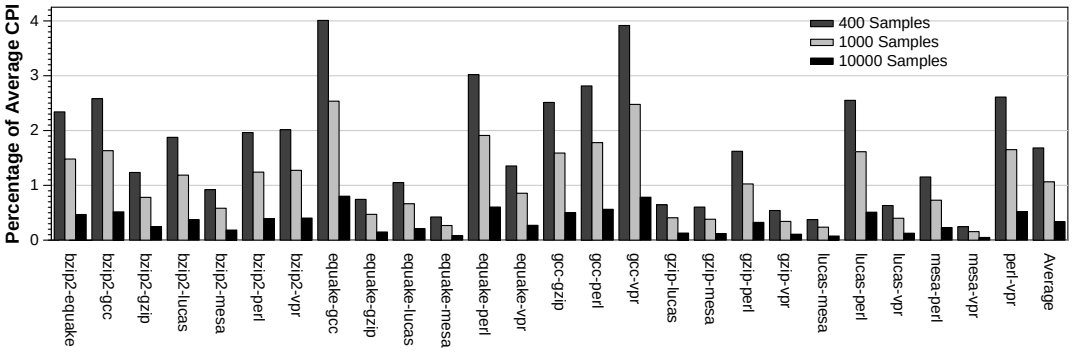

Figure V.8: Confidence intervals for varying numbers of random samples.

approach, we first fill our static co-phase matrix with estimated CPIs as described earlier.

For each sample CPI, a starting location is randomly chosen separately for each program. Then a single run is performed of the static co-phase method of those starting points for 1 billion instructions using the static co-phase matrix approach in Section IV.B.11. This is repeated as we collect samples for many possible starting point combinations. The graphs show the effects on estimated All Combination CPI as the number of samples is increased. As the results show, the accumulated samples eventually converge to an All Combination CPI, however the techniques converge at different rates.

For the results, we did this 1024 times and calculated the cumulative average at each step. The estimates for most benchmark pairs converged after about 400 samples. Given a desired confidence level, it is possible to sample until that confidence level is achieved and report error bounds along with the estimate. We examined improving the convergence rate of the random sampling method using stratified random sampling. Each program was divided into four parts, giving 16 partitions per benchmark pair. We sampled equally from each of the 16 partitions of possible starting offsets. This had the effect of decreasing the variation when the number of samples was small (up to 200), but did not

provide much help for greater numbers of samples. Note that plotted lines in Figure V.7 are less noisy because points were plotted every 16 samples (due to stratification); only the variation in magnitude is important.

Random sampling allows us to adjust the number of samples taken so that the error due to the limited number of samples has a high probability of being within a small margin. Figure V.8 shows, with 95% confidence, a bound on the difference between the actual average CPI of all possible starting combinations and our estimated CPI when using 400, 1000 and 10000 samples. The difference between the actual and estimated CPI is represented as an absolute percentage from the estimated CPI in Figure V.8.

The size of the confidence intervals is dependent upon the natural variability of the benchmark combinations, the number of samples and the confidence desired. Although random sampling appears to converge after 400 samples for most benchmark pairs, the size of the confidence interval is still quite high. The results show that using 1000 samples and 95% confidence, the real CPI should be within just 1% of the estimated average CPI. All confidence intervals are under 0.8% of average CPI when using 10000 samples (just 0.34% on average). When using 10000 random samples, less than five minutes of static co-phase matrix simulation time per benchmark pair is sufficient to calculate the AC CPI.

V.B.3 Reducing Co-phases Using PCA

We now present our methodology for building a reduced but representative multiprogrammed workload for driving the simulation of multithreaded processors. The workload that we start with is a set of benchmarks that the computer architect considers a viable set of benchmarks. The set of benchmarks that we consider in the remainder of this chapter is the entire set of reference inputs for the SPEC CPU 2000 benchmark suite. The overall goal of the method-

ology proposed in here is to reduce this workload into a workload that is viable for simulation purposes while being representative for the multiprogrammed behavior that is to be expected with the original workload. Our workload reduction methodology consists of four steps.

1. The first step determines the prominent phase behaviors for each of the benchmarks in the original workload. The prominent phase behaviors are represented by simulation points — a simulation point is represented by a position in the dynamic instruction stream where the phase behavior starts. In practice, a simulation point is stored on disk using checkpoints [55]. The number of simulation points is limited to a few per benchmark. The goal is for each simulation point to exhibit homogeneous program behavior.

2. For each of these simulation points, we then measure a number of microarchitecture-independent characteristics. This is done very efficiently through profiling.

3. Next, principal components analysis (PCA) [24] is applied to a data set that represents all of the simulation points. The goal of principal components analysis is to identify similar phases based on their microarchitecture-independent behavior.

4. In the final step, cluster analysis (CA) is applied to the results of the PCA in order to find groups or clusters of *co-phases* (combinations of simulation points) that exhibit similar microarchitecture-independent behavior. A representative co-phase is then chosen for each cluster and each co-phase is represented as a weighted sum of neighboring representative co-phases. The set of representative co-phases and sum of associated weights then constitute the reduced workload.

We now discuss each of these four steps in greater detail in the following subsections.

V.B.4 Finding Homogeneous Intervals

The first issue we need to address is how we end our multiprogram simulation samples, since during simulation each program being simulated in a multiprogrammed workload will have different rates of progress yet we do not want the nature of the workload to change due to a change in relative rates of progress when we use a different microarchitecture configuration. Inside the 50M-instruction intervals there will be small-scale repetitive behavior from loops. If the repetitive behavior is small enough then it is fine to simulate only 10M instructions from one program's 50M interval while simulating 40M instructions from the other interval, even though another microarchitectural configuration might allow them to progress at the same rate. On the other hand, a 15M-instruction loop with different behavior in the last 5M instructions would not provide a workload that is consistent between these two configurations.

To create homogeneous intervals we perform SimPoint phase analysis at a smaller interval size than the 50M-instruction interval granularity to verify whether the phase behavior within the 50M instructions is constant. In most cases, especially for floating-point benchmarks, all of the frequent phases contain instances that have consistent behavior for at least 50M instructions.

V.B.5 Microarchitecture-Independent Characteristics

In order to be able to identify similarity across simulation points, we consider microarchitecture-independent characteristics measured over these simulation points. As mentioned before, the reason we consider microarchitecture-independent characteristics is that the workload analysis needs to be done only

once so that its results can be used multiple times for estimating the performance of a collection of processor configurations. This is important since we want to run our analysis once on a benchmark suite and use the co-phases found across all the different architecture configurations during design space explorations. Table V.1 summarizes the microarchitecture-independent characteristics that we use in the remainder of this chapter, which we now describe.

The range of microarchitecture-independent characteristics is fairly broad in order to cover all major program behaviors such as instruction mix, inherent ILP, working set sizes, memory strides, branch predictability, *etc.* The results given our evaluation confirm that this set of characteristics is indeed broad enough for accurately characterizing cross-program and cross-input similarity. We include the following characteristics:

Instruction mix. We include the percentage of loads, stores, control transfers, arithmetic operations, integer multiplies and floating-point operations.

ILP. In order to quantify the amount of instruction-level parallelism (ILP), we consider an idealized out-of-order processor model in which everything is idealized or unlimited except for the window size. We measure for a given window size over a set of 32, 64, 128 and 256 in-flight instructions how many independent instructions there are within the current window.

Register traffic characteristics. We collect a number of characteristics concerning registers [16]. Our first characteristic is the average number of input operands to an instruction. Our second characteristic is the average degree of use, or the average number of times a register instance is consumed (register read) since its production (register write). The third set of characteristics concerns the register dependency distance. The register dependency distance is defined as the number of dynamic instructions between writing a register and reading it.

Table V.1: Microarchitecture-independent characteristics.

Category	No.	Characteristic
Instruction Mix	1	% loads
	2	% stores
	3	% control transfers
	4	% integer operations
	5	% floating-point operations
	6	% no-operations
	7	% software prefetch operations
ILP	8	32-entry window
	9	64-entry window
	10	128-entry window
	11	256-entry window
Register Traffic	12	avg. number of input operands
	13	avg. degree of use
	14	prob. register dependence = 1
	15	prob. register dependence $\leq$ 2
	16	prob. register dependence $\leq$ 4
	17	prob. register dependence $\leq$ 8
	18	prob. register dependence $\leq$ 16
	19	prob. register dependence $\leq$ 32
	20	prob. register dependence $\leq$ 64
Working Set Size	21	I-stream at the 32B block level
	22	I-stream at the 4KB page level
	23	D-stream at the 32B block level
	24	D-stream at the 4KB-page level
Data Stream Strides	25	prob. local load stride = 0
	26	prob. local load stride $\leq$ 8
	27	prob. local load stride $\leq$ 64
	28	prob. local load stride $\leq$ 512
	29	prob. local load stride $\leq$ 4096
	30	prob. local store stride = 0
	31	prob. local store stride $\leq$ 8
	32	prob. local store stride $\leq$ 64
	33	prob. local store stride $\leq$ 512
	34	prob. local store stride $\leq$ 4096
	35	prob. global load stride = 0
	36	prob. global load stride $\leq$ 8
	37	prob. global load stride $\leq$ 64
	38	prob. global load stride $\leq$ 512
	39	prob. global load stride $\leq$ 4096
	40	prob. global store stride = 0
	41	prob. global store stride $\leq$ 8
	42	prob. global store stride $\leq$ 64
	43	prob. global store stride $\leq$ 512
	44	prob. global store stride $\leq$ 4096
Branch Predictability	45	GAg PPM predictor
	46	PAg PPM predictor
	47	GAs PPM predictor
	48	PAs PPM predictor

Working set. We characterize the working set size of the instruction and data stream. For each interval, we count how many unique 32-byte blocks were touched and how many unique 4KB pages were touched for both instruction and data accesses.

Data stream strides. The data stream is characterized with respect to local and global data strides [29]. A global stride is defined as the difference in the data memory addresses between temporally adjacent memory accesses. A local stride is defined identically except that both memory accesses come from a single instruction—this is done by tracking memory addresses for each memory operation. When computing the data stream strides we make a distinction between loads and stores.

Branch predictability. The final characteristic we want to capture is branch behavior. The most important aspect would be how predictable the branches are for a given interval of execution. In order to capture branch predictability in a microarchitecture-independent manner we used the Prediction by Partial Matching (PPM) predictor proposed by Chen *et al.* [6], which is a universal compression/prediction technique.

A PPM predictor is built on the notion of a Markov predictor. A Markov predictor of order k predicts the next branch outcome based upon k preceding branch outcomes. Each entry in the Markov predictor records the number of next branch outcomes for the given history. To predict the next branch outcome, the Markov predictor outputs the most likely branch direction for the given k-bit history. An m-order PPM predictor consists of $(m + 1)$ Markov predictors of orders 0 up to m. The PPM predictor uses the m-bit history to index the m^{th}-order Markov predictor. If the search succeeds, *i.e.* the history of branch outcomes occurred previously, the PPM predictor outputs the prediction by the mth order Markov predictor. If the search does not succeed, the PPM predictor

uses the $(m-1)$-bit history to index the $(m-1)^{\text{th}}$-order Markov predictor. In case the search misses again, the PPM predictor indexes the $(m-2)^{\text{th}}$-order Markov predictor, *etc.* Updating the PPM predictor is done by updating the Markov predictor that makes the prediction and all its higher order Markov predictors. In this chapter, we consider four variations of the PPM predictor: GAg, PAg, GAs and PAs. 'G' means global branch history whereas 'P' stands for per-address or local branch history; 'g' means one global predictor table shared by all branches and 's' means separate tables per branch. We want to emphasize that these metrics for computing the branch predictability are microarchitecture-independent. The reason is that the PPM predictor is to be viewed as a theoretical basis for branch prediction rather than an actual predictor that is to be built in hardware.

V.B.6 Workload Characterization

The collected microarchitecture-independent data is useful for research purposes with minimal further analysis, as PCA reveals important quantitative, microarchitecture-independent, properties of each interval and simple statistical techniques reveal the nature of our workload. For each characteristic we can compare the collected data for all intervals, finding the highest and lowest values as well as grouping the intervals into quintiles: very low, low, medium, high and very high. Determining mean values and standard deviation is also possible but might not be useful for many characteristics since probabilities are unlikely to be normally distributed. Once we have classified all intervals according to their characteristics there are three important things that we can do.

First, we can evaluate a processor using combinations of the extreme behaviors. Researchers can pick intervals with properties appropriate for their experiments, just as entire benchmarks known to have particular execution properties are used for single-threaded experiments. For example, we can run only

intervals with very low ILP or very high probability of control transfer and low probabilities for branch predictors. The working set size and data stream predictors can be used to test cache configurations. The highest and lowest values for a characteristic indicate intervals suitable for studying the limits of processors without resorting to a synthetic workload. For example, our collection of 50M-instruction intervals for SPEC CPU 2000 includes one that is mostly stores with no loads and another that is mostly loads with no stores.

Second, we can identify the most average intervals, those that are never categorized as very high nor very low and have the most medium categorizations. Collections of these intervals form a baseline performance model when investigating the effects of changing workloads on a fixed machine configuration.

Third, after running a number of interval combinations and finding that some of them do poorly, we can correlate performance with quintiles for each characteristic. It is easy to determine which characteristics are independent of problems and focus on the remaining ones. In many cases the characteristics will suggest which resources, such as functional units or prefetchers, need to be improved to handle the problematic workloads.

V.B.7 Principal Components Analysis

Principal components analysis (PCA) [24] is a statistical data analysis technique that presents a different view on a given data set. The two most important features of PCA are that (*i*) PCA is a data reduction technique that reduces the dimensionality of a data set and (*ii*) PCA removes correlation from the data set. Both features are important to increase the understandability of the data set. For one, analyzing a q-dimensional space is obviously easier than analyzing a p-dimensional space in case $q \ll p$. Second, analyzing correlated data might give a distorted view; non-correlated data does not have that problem. The

reason is that a distance measure in a correlated space gives too much weight to correlated variables — these correlated variables result from the same underlying program characteristic; the underlying characteristic would thus have too much weight in the overall distance measure.

The input to PCA is a matrix in which the rows are the *cases* and the columns are the *variables*. In this chapter, each row represents a single 50-million instruction interval. The columns represent the 48 microarchitecture-independent characteristics presented in the previous subsection for each of the phases in a co-phase.

PCA computes new variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA transforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the properties (i) $Var[Z_1] \geq Var[Z_2] \geq \ldots \geq Var[Z_p]$ — this means Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ — this means there is no information overlap between the principal components. Note that the total variance in the data (variables) remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$. In this chapter, X_i is the ith microarchitecture-independent characteristic; Z_i then is the ith principal component after PCA. $Var[X_i]$ is the variance of the original microarchitecture-independent characteristic X_i computed over all intervals. Likewise, $Var[Z_i]$ is the variance of the principal component Z_i over all intervals.

As stated in the first property in the previous paragraph, some of the principal components will have a high variance while others will have a small variance. By removing the principal components with the lowest variance from the analysis, we can reduce the dimensionality of the data while controlling the amount of information that is thrown away.

We retain q principal components which is a significant information reduction since $q \ll p$ in most cases. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $(\sum_{i=1}^{q} Var[Z_i])/(\sum_{i=1}^{p} Var[X_i])$ accounted for by these q principal components. For example, criteria such as '70%, 80% or 90% of the total variance should be explained by the retained principal components' could be used for data reduction. An alternative criterion is to retain all principal components for which the individual retained principal component explains a fraction of the total variance that is at least as large as the minimum variance of the original variables.

By examining the most important q principal components, which are linear combinations of the original variables ($Z_i = \sum_{j=1}^{p} a_{ij} X_j, i = 1, \ldots, q$), meaningful interpretations can be given to these principal components in terms of the original microarchitecture-independent characteristics. A coefficient a_{ij} that is close to +1 or -1 implies a strong impact of the original characteristic X_j on the principal component Z_i. A coefficient a_{ij} that is close to 0 on the other hand, implies no impact.

In principal components analysis, one can either work with normalized or non-normalized data — the data is normalized when the mean of each variable is zero and its variance is one. In the case of non-normalized data, a higher weight is given in the analysis to variables with a higher variance. In our experiments, we have used normalized data because of our heterogeneous data; *e.g.*, the variance of the ILP is orders of magnitude larger than the variance of the instruction mix.

The output obtained from PCA is a matrix in which the rows are the 50M phases and the columns are the retained principal components. Before we proceed to the next step we make sure we normalize the principal components, *i.e.*, we rescale the principal components to unit variance. The reason is that a non-unit variance of a principal component is a consequence of the correlation as

observed in the original data set. And since our next step in the data analysis uses a distance measure to compute the similarity between cases, we make sure correlation does not give a higher weight to correlated variables.

V.B.8 Cluster Analysis

The next step in our workload reduction methodology is to perform cluster analysis (CA) [24] on co-phases. There exist two commonly used strategies for applying cluster analysis, namely linkage clustering and k-means clustering. Since k-means clustering is less compute-intensive than linkage clustering and a component of SimPoint, we use k-means in this chapter.

The input to the cluster analysis is a matrix in which the rows are all possible co-phases and the columns are the retained principal components for each phase in the co-phase. Cluster analysis thus finds a number of groups or clusters of co-phases that exhibit similar microarchitecture-independent behavior. We only include distinct co-phases in the matrix: if A and B are phases, then co-phases AB and BA are considered identical.

Our definition of distinct co-phases causes a problem for clustering. If A and D are phases with similar properties and so are B and C, then we would like the co-phases AB and CD to be similar. The normal Euclidean distance metric would consider the rows of statistics representing these co-phases to be far apart unless all of the phases were similar. The co-phases AB and DC would be close together, however. We avoid this problem by using a different distance metric. In this metric, the distance between two co-phases is the minimum of the Euclidean distances between the first co-phase and the two orderings of the second one.

Some of the remaining clusters have very low weight. To decrease the number of simulations required, we can eliminate clusters with a weight below a given threshold, such as 0.5% (reweighting all remaining clusters accordingly).

This can significantly decrease the number of co-simulation points with negligible effect on the accuracy of collected statistics.

V.B.9 Interpolation of Cluster Centers

In standard SimPoint, the co-phase that is closest to each cluster's centroid is called the representative co-phase. The weight assigned to this representative co-phase, referred to as the *co-simulation point* is the sum of the weights of the co-phases that are members of the given cluster divided by the total weight of all co-phases. Only the representative simulation points need to be simulated when we estimate performance numbers.

To improve our accuracy without increasing simulation time, we observe that points that are between other points should have in-between performance. In a Euclidean metric space we could choose cluster centers that form a convex hull around the target point and use geometry to determine the weight of each selected center. This is much more challenging in our metric, so we use a simpler scheme. Each point is computed as a weighted average of its n nearest neighbors. If the closest neighbor is at distance d_0, then each point has relative weight $e^{-c\frac{d}{d_0}}$. Appropriate choices for n and c depend upon both the number of cluster centers and each other — a large c will compensate for an overly-large n by discounting faraway neighbors; a small c requires a small n so that faraway points are not included).

V.B.10 Weighting Average Throughput

To simplify comparisons between techniques to reduce the number co-simulation points, we propose a single weighted average throughput metric. We consider two types of weights, the weight of a pair of benchmarks and the weight of co-phases for each pair of benchmarks.

Each benchmark consists of a program and its input. Programs have from one to five inputs, but a program with many inputs is not necessarily more significant than a program with a single input. Thus, we consider each program equally important, as is every pairing of programs. Each input is equally important as any other for the same program. In this scheme, the weight of `lucas` and `mesa` is 25 times that of `gcc-166` and `gzip-program` since `gcc` and `gzip` each have 5 inputs.

For a given pair of benchmarks, we must subdivide the weights between co-phases. Unlike programs and inputs, the co-phases clearly should have distinct weights because the phases that compose them are known to have particular weights. Some phases represent less than 5% of a benchmark whereas others represent over 90% of benchmark. The weight that we give to a co-phase is equal to the product of the weights of the constituent phases. Thus if one phase represents 20% of a benchmark and the other 30%, the weight of the co-phase is 6% that of the pair of benchmarks.

When the number of threads is large there may be too many co-phases to estimate their performance efficiently. Random sampling allows the weighted average to be estimated efficiently at any desired level of accuracy. Our analysis procedures allow the average weighted throughput to be estimated using detailed simulation of only a small number co-simulation points.

V.B.11 Baseline Simulator

We use the M5 simulator [4] from the University of Michigan, which is based on SimpleScalar3.0c [5] as our SMT simulation environment. The configurations used for this simulator are shown in Table V.B.11. It is configured to support an intensive multithreaded workload; hence the abundant reorder buffer and processor width. The memory hierarchy is based on current-generation pro-

Table V.2: SMT processor configurations.

I-Cache	32kB 2-way set-associative, 64-byte blocks, 1-cycle latency *or* 64kB 2-way set-associative, 64-byte blocks, 1-cycle latency
D-Cache	32kB 8-way set-associative, 64-byte blocks, 3-cycle latency *or* 64kB 8-way set-associative, 64-byte blocks, 3-cycle latency
Unified L2	1 MB 8-way set-associative, 128-byte blocks, 10-cycle latency *or* 4 MB 16-way set-associative, 128-byte blocks, 14-cycle latency
Memory	250-cycle latency
Branch Pred	21264-style hybrid predictor with 13-bit global history indexing a 8k-entry global PHT and 8k-entry choice table; 2k 11-bit local history entries indexing a 2k-entry local PHT A: 4kB, 4-way set-associative BTB; 3-cycle misprediction recovery *or* B: 4kB, 2-way set-associative BTB; 2-cycle misprediction recovery
OOO Issue	out-of-order issue, 256-entry re-order buffer
Width	8 instructions per cycle (Fetch, Decode, Issue and Commit)
Func Units	6 Integer, 2 Integer Multiply, 4 FP Add, 2 FP Multiply

cessors. For the L1 caches, unified L2 cache and branch predictor we considered two design points each, for eight possible combinations. We simulated SPEC CPU2000 benchmarks compiled for the Alpha ISA.

Each co-phase was executed until a combined 50M instructions were committed by both threads. Since our target workload (all co-phases) is constant and each phase is homogeneous, we calculate performance using throughput in instructions per cycle. Due to the long simulation period, warmup effects correspond to less than 0.5% variation in throughput. Nonetheless, we ignore the first 5M combined instructions to remove error due to warmup effects.

V.B.12 Cluster and Principal Components Analysis

We analyzed the benchmarks and microarchitecture-independent co-phase features using SimPoint 3.0 [19]. When analyzing benchmarks with SimPoint we found up to 10 phases per benchmark. We selected an average of 5 phases per program by removing phases that corresponded to less than 2.5% of program execution.

The microarchitecture-independent analysis was performed using a modified version of SimpleScalar. We analyzed each of the 50M-instruction simulation points found in the previous step.

From the PCA step we selected the 4 most-significant dimensions, which were sufficient to explain over 44% of the variance. Thus clustered 8-dimensional data. Increasing the number of dimensions used leads to poorer cluster analysis as clustering treats all of the dimensions as equally significant — this leads to the curse of dimensionality problem.

V.B.13 Homogeneous Intervals

The accuracy of our simulations depends on homogeneous behavior within each 50M-instruction interval. If the pattern of execution for one program deviated significantly near the end of the simulation interval, this would affect simulations that execute the different code, but some simulations might make faster progress with the second program and thus never execute the different code. It would be misleading to compare the results of the two experiments. Thus, we need to verify that the intervals contain homogeneous behavior. To do this, we examine the execution of all co-phases on our baseline processor and observe the effects of varying the length of simulation between 45M and 50M instructions, in 0.5M instruction increments. At each increment we compare the IPC with the IPC prior to the increment and determine the relative difference

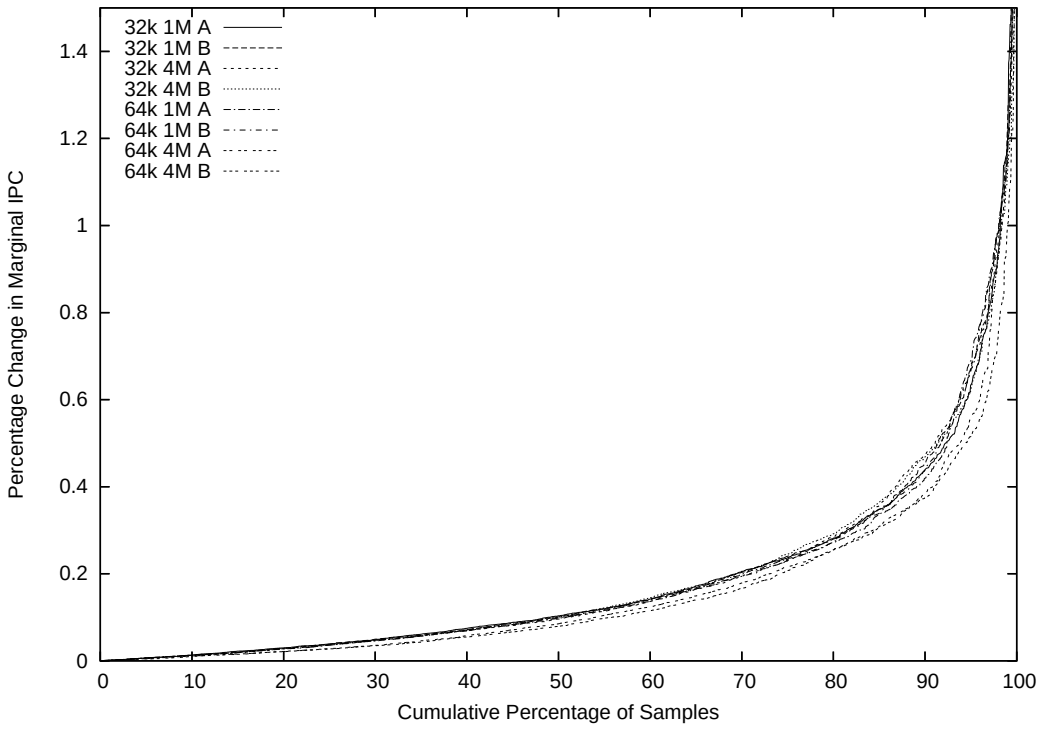

Figure V.9: Cumulative distributive function for marginal change in IPC.

caused by the slightly longer execution. We plot these values in Figure V.9 as a cumulative distributive function (CDF) for eight microarchitectures. For 80% of samples the variation in throughput is at most 0.3% and less than 1% of samples cause a variation of more than 1.2%. Thus we can expect that our simulations will provide stable, reliable results that are not sensitive to the exact point at which simulation terminates. Furthermore, the error rate is not particularly sensitive to the machine configuration.

Variation in the middle of a simulation could also lead to incomparable executions provided that both programs have significant variation, as we demonstrated in our previous work [56, 59]. For the homogeneous intervals in this chapter, the degree of variation in the middle of the intervals is similar to that at the end of the execution intervals. The use of 50M-instruction intervals ensures

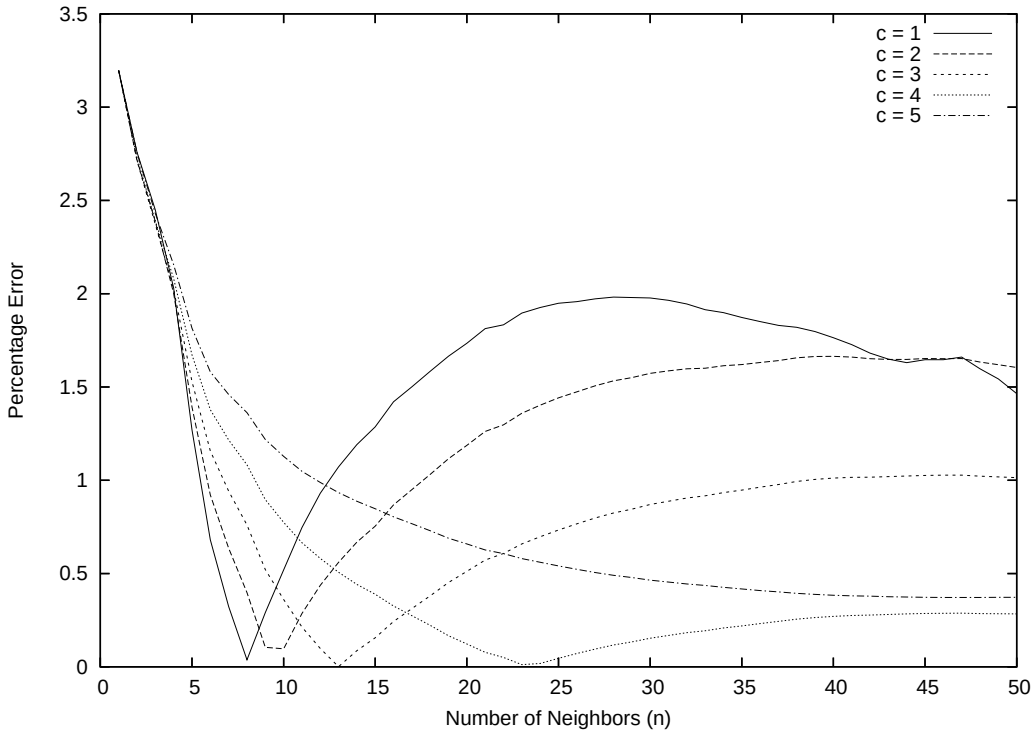

Figure V.10: Error using different interpolation parameters (configuration 32k 4M A).

that the natural fine-grain program variation is insignificant on the scale that we sample.

V.B.14 Interpolation

In Figure V.10 we examine the effects on estimating throughput that changing the parameters to the interpolation algorithm has, as described in Section V.B.9. For each combination of parameters we use 50 representative points. We use one line for each choice for constant c. The x-axis is the number of neighbors used to compute throughput, n. Two values of n are of particular note. When $n = 1$ the algorithm is equivalent to the standard SimPoint algorithm that selects a single representative simulation point (thus c has no effect). Al-

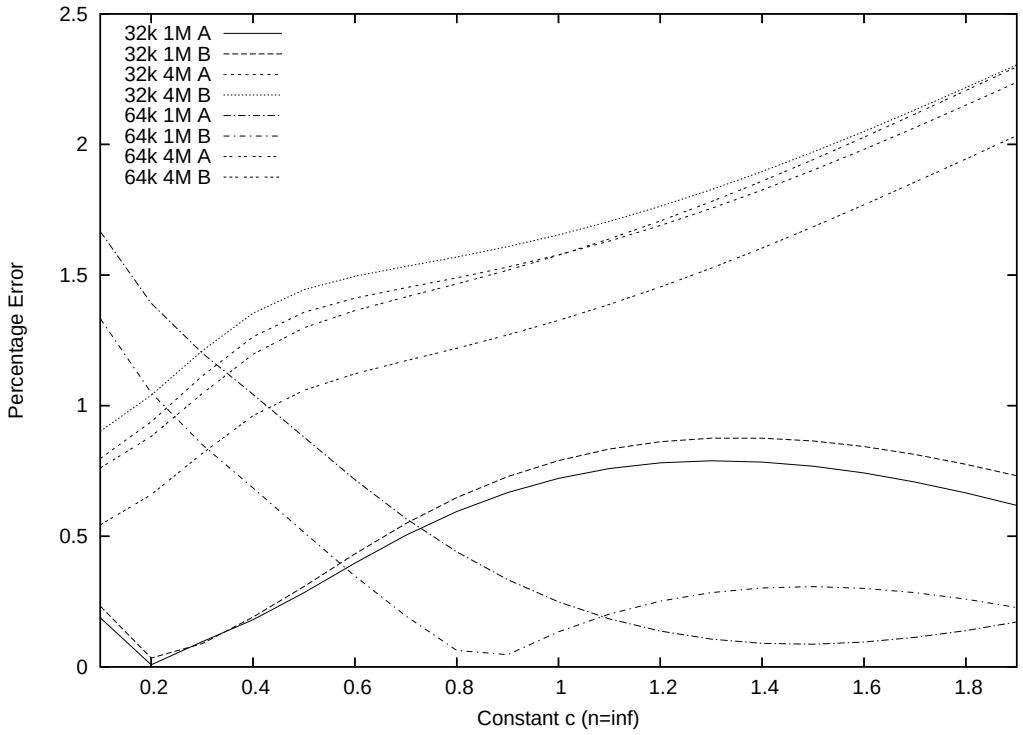

Figure V.11: Error varying c using all configurations.

though this method is reasonably accurate (3.2% error), it performs worse than any other combination of parameters. At $n = 50$, all representative points are used. Their weights are dependent upon their distances and c. The larger values of c lead to more accurate results because they give negligible weight to distant representative points. Small values of c and n combine to get excellent results but the sharp inflection points indicate the need for fine-tuning. All methods give excellent results (under 2% relative error) as long as at least 5 neighbors are used, but using all the points is the most robust option.

In Figure V.11 we examine the effects on error of using different values of c with all eight microarchitecture configurations. We see that the error rates are low in all cases, but different depending on machine configuration. The configurations with 4M L2 caches have similar changes in error rates for all c.

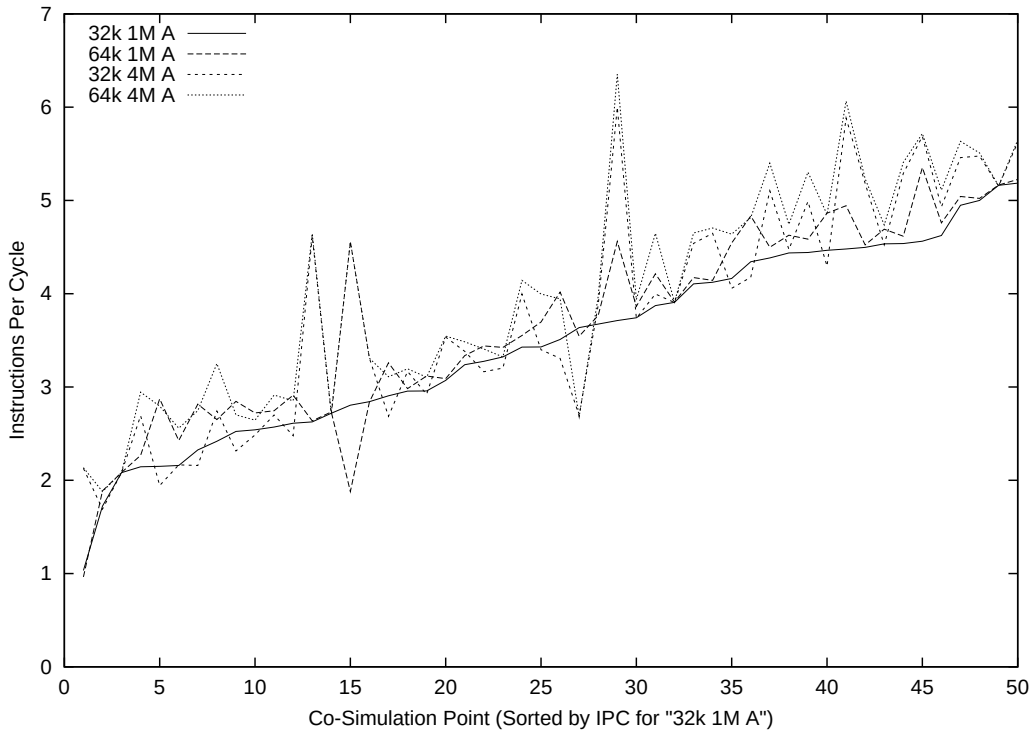

Figure V.12: Effects of configuration choice on co-simulation point performance.

The remaining configurations are also distinguishable by L1 cache size. Since the error rates are low, any value of c in this range can be used to accurately compare different microarchitectural configurations.

V.B.15 Summarizing Benchmark Suite Performance

In Figure V.12 we examine the sampled performance for each of the 50 co-simulation points, looking at the four microarchitectural configurations that use branch predictor A. (The results for branch predictor B are too similar to A to show at the same time.) The co-simulation points are ordered according to the performance of the configuration with 32k L1 caches and 4M of L2 cache. For the configurations with 4M of cache, it is never worse to use 64k L1 caches than 32k caches. For all other pairs of configurations there are points favoring either

configuration. In particular, it is hard to determine whether the configuration with 32k L1 caches and a 4M L2 cache is better than the configuration with 64k L1 caches and a 1M L2 cache. This reinforces the point that it is important to look at many diverse program interactions and to weight them appropriately.

The co-simulation point weights used for two sets of interpolation parameters can be seen in Figure V.13. Most points have similar weights using both sets of parameters despite the significant differences in interpolation techniques, so results using either set of parameters should be similar. The ratio of weights between the least and most significant co-simulations points is roughly 200:1. These weights are applied to CPI samples, which can range in theory from about 0.125 (fetch constrained) to 250 (long sequences of dependent loads) on our test architecture so the least-weighted points could theoretically contribute more to CPI than the greatest-weighted ones. The heaviest point weights are not so great as to dominate the benchmark suite.

In Figure V.14 we show the actual overall performance numbers that were calculated using all eight microarchitectural configurations and both sets of interpolation parameters. The greatest difference in estimated performance is just 2.5%, which is in line with our error calculations in Figure V.10 and Figure V.11. Both interpolation parameters order the eight configurations identically. All else being equal, the B branch predictor has a minuscule performance advantage over the A branch predictor. We can see that using the 4M L2 cache is the most important performance improvement that can be made and that 64k L1 caches are better than 32k L1 caches. In Figure V.12 it is difficult to tell if 32k L1 caches and a 4M L2 cache is better than 64k L1 caches and a 1M L2 cache, but the weighted average provides a balanced comparison between the configurations.

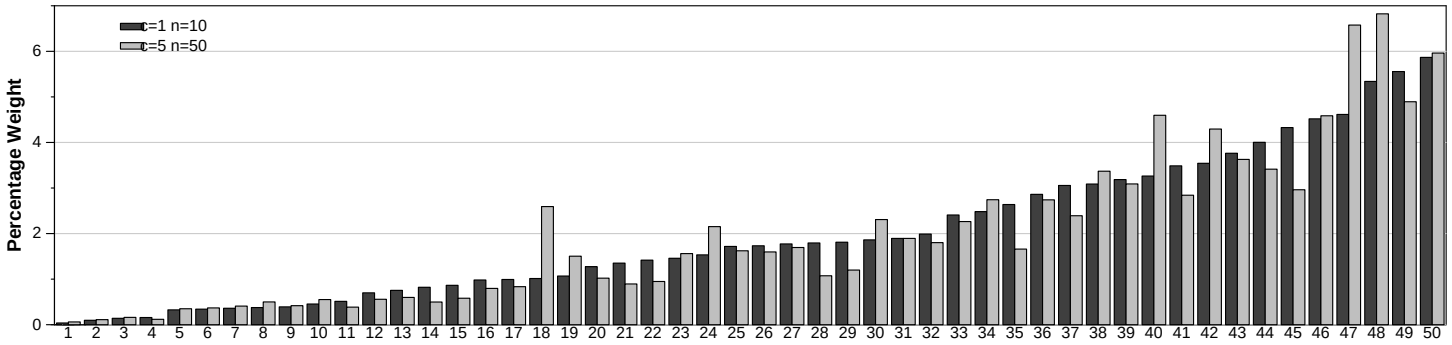

Figure V.13: Weights used for each co-simulation point using two interpolation parameters.

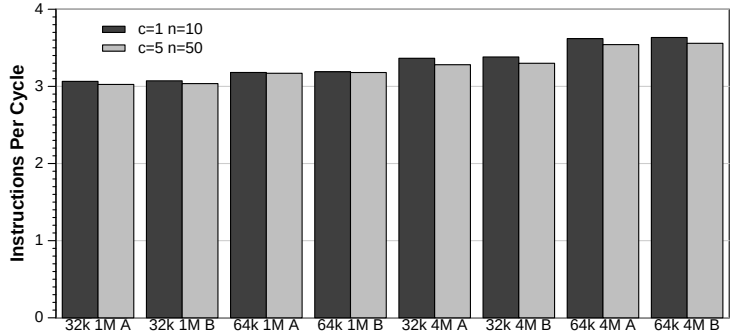

Figure V.14: Overall average IPC using two interpolation parameters.

V.B.16 Clustering Using More Than Two Threads

Ultimately, we would like our clustering technique to scale to large numbers of cores and threads so that it can handle processors such as the POWER5 and Niagara2. There are two problems to consider. First, there is the larger number of threads to consider; looking at all permutations of threads could have exponential cost. Second, interchanging threads between contexts, cores on a die, chips in a package and packages on a motherboard are all different in effect.

In the simplest case of a single multithreaded core or several single-threaded cores, we just consider all permutations of threads to be identical. For dies with multiple multithreaded cores, each group of threads on a core may be reordered and the cores may be reordered without making a distinctly different co-phase, but intermixing the threads of two cores will produce a distinct co-phase. It is easy to tell if two co-phases are distinct. Sort the threads in each core and the sort the cores in lexicographic order. The resulting structures will be identical if the co-phases are equivalent (not distinct). We can easily generate distinct co-phases randomly or exhaustively using this canonical form. The process extends naturally to any hierarchy of symmetrical components.

This definition of distinct co-phases minimizes the number of inputs to the clustering and results in the best possible results for a given number of clusters. Unfortunately, it results in different simulation points for different core and SMT context configurations even when the total number of threads are the same. If this is undesirable, then the clustering can be done using a chip layout that requires a superset of the reordering restrictions of all of the relevant chip configurations. For example, 1×8, 8×1, 4×2 and 2×4 chip layouts can be approximated by a $2 \times 2 \times 2$ chip layout. This ensures that if two threads are distinct in one of the original configurations then they will be distinct in the new configuration.

Calculating this distance metric naïvely would be an expensive operation because it requires $n!$ distance computations for n threads. For example, for 8 threads, we would need to compute $8! = 40320$ distances. Fortunately, this number assumes a lot of repeated work. Each time the distance between a pair of phases is calculated there are 4 dimensions to consider (each a subtraction followed by a multiplication). There are only 64 pairs of phases, so the distances between the phases in each pair need only be calculated once. Each of the 8! distances between co-phases is reduced to the summation of 8 table lookups, and an obvious stack-based algorithm will average under 3 table lookups and additions per distance calculated. These optimizations alone reduces the slowdown to a factor of about 1700 on a machine with fast multiplies (better on other machines). If each core has two threads the normal distance metric will take twice as long. The 64-entry table will take four times as long to generate (twice as many columns and two orders), but the 8! part of the algorithm will run at the same speed, so the slowdown is nearly halved. Memoization can reduce the slowdown to 30–40 times for both 8-core configurations using a table with only 2^8 entries. Each entry in the table represents the minimum cost of mapping a subset of the 8 cores in the permuted co-phase to the initial cores of the other co-phase.

The number of co-phases for an 8-core SMT machine would cause the clustering algorithm a much greater slowdown than the distance metric — there are about 10^{29} distinct co-phases that can be formed from our 50-million instruction intervals. When we use more than two cores or threads we use a random sample of co-phases. Weighting the probability of selection of a co-phase according to the product of the weights of the component phases ensures that the chosen centers will be near to the co-phases with greatest weight.

Using these optimizations, a careful implementation of our distance metric and random sampling of co-phases, we can analyze multithreaded workloads

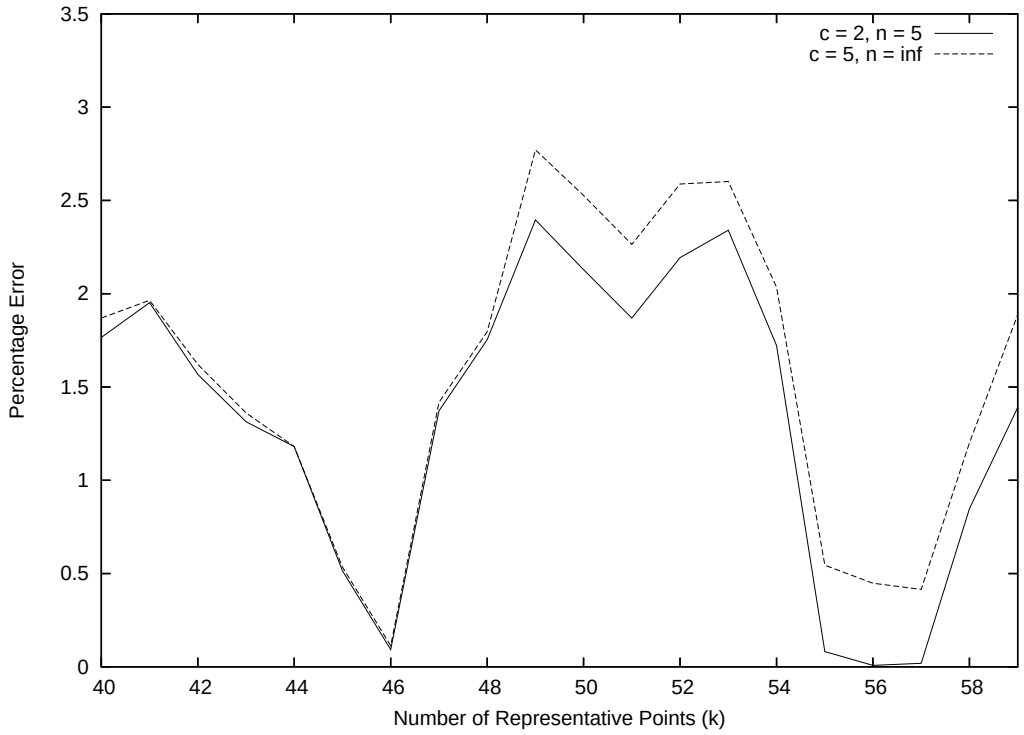

Figure V.15: Error using different numbers of randomly chosen representative points (configuration 32k 4M A).

consisting of large numbers of threads. This analysis will allow us to simulate just a few representative instances, vastly reducing analysis time for modern computer architectures.

V.B.17 Random Representative Points

We also investigated using random selection of representative points rather than cluster centers. Without interpolation, the results were significantly worse than using clustering. Interpolation, however, leads to similar results whether centers are chosen randomly or by using clustering. The main difference that we found were slightly higher error rates and a preference for slightly fewer neighbors. As we can see in Figure V.15, we consistently get at most 2.5%

error when using varying numbers of randomly chosen cluster centers. Since we can get accurate results despite randomly selecting centers, we can scale the algorithm to large numbers of cores and threads. Randomly sampling the set of possible co-phases and clustering the results should give good results even though the 'best' cluster centers might not be in the sample sets. Clustering ideally uses thousands of iterations of cluster center movement with distance comparisons to all points in every step. Should our distance metric be too expensive for huge numbers of cores or threads, the random center results suggest that we could reasonably reduce the number of iterations during clustering to compensate for distance metric costs, or even eliminate clustering altogether.

V.C Summary

In this chapter we showed that it is important to consider multiple starting points in order to obtain a reliable multithreaded performance number. Moreover, we presented an efficient multithreaded simulation methodology for achieving this. By building up a co-phase matrix that summarizes the performance of all the co-phase executions, we are able to quickly estimate the average performance for all possible starting points. This is done by sampling over all possible starting points and by analytically simulating those randomly selected starting points over the co-phase matrix. Due to the use of the static co-phase method we were able to show an average sample collection bias of under 1.6%.

We evaluated two sampling approaches, random sampling and stratified random sampling. We observed that both sampling strategies resulted in around 400 starting points that need to be simulated in order to get stable performance estimates, but using 1000 or more samples allows strong confidence bounds. Since each sample can be collected in a fraction of a second, and a confidence interval on sampling error below 0.8% can be obtained in just a few minutes. The end result

is a multithreaded simulation methodology that estimates average multithreaded performance over all combinations of starting points in the order of minutes once the co-phase matrix is populated with samples.

Additionally, architecture studies of multithreaded processor need to balance the performance requirements of every combination of benchmarks. Simulating all of the benchmark combinations is excessively time-consuming, even when using sampling techniques such as the co-phase matrix. We demonstrate a technique for analyzing a benchmark suite and finding all of the distinct co-phase behaviors that can occur when pairs of benchmarks run together. By clustering the co-phases behaviors we are able to find representative *co-simulation points* that can be simulated as substitute for simulating all of the co-phases. We demonstrate that less than 50 co-simulation points provide results differing by less than 2.5% from simulating all co-phases.

This set of co-simulation points can be used to compare the performance of different microarchitectural configurations by executing 2.5 billion instructions per configuration. Our simulation point selection technique simplifies the simulation procedure by ensuring that each co-simulation point has homogeneous behavior.

V.D Acknowledgements

This chapter contains material from *Considering All Starting Points for a Simultaneous Multithreading Simulation Methodology* [56], in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper.

creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This chapter contains material from *Representative Multiprogram Workloads for Multithreaded Processor Simulation* [58], in *IEEE International Symposium on Workload Characterization (IISWC)*, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder. The dissertation author was the primary investigator and author of this paper. Portions of these chapters are ©2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

VI

Conclusion

This dissertation demonstrates that simulation must be conducted carefully, with care taken for simulator warmup, representing program phase behavior during multithreaded simulation and the variation in multithreaded performance that comes from starting simulation at different points in benchmarks.

More significantly, we provide solutions for these problems that are not only accurate, but efficient – both in time and disk resources. We showed that storing simulator state to disk could replace lengthy simulator warming and optimized the size of checkpoints. As a result, researchers can quickly evaluate processor performance from a wide variety of checkpoints without sacrificing either accuracy or disk space; this is desirable for any sampled simulation environment.

A small number of checkpoints per benchmark can be used to model all of the different interactions between benchmarks running on a simulated multithreaded processor. The results of simulating these interactions forms the co-phase matrix. Our co-phase method of approximating multithreaded execution starts at any given point of co-execution and joining together per-program phase information using the co-phase matrix to instantly predict subsequent execution.

The ability of the co-phase method to instantly evaluate execution from any starting offset allows us to efficiently randomly sample the space of all possible

simulation starting offsets. This allows us to summarize the average performance of a pair of programs without missing any interesting interactions.

Finally, we have begun to use principle components and clustering analysis to find similarities between phases in different benchmarks. Currently, this allows us to use a modest amount of simulation time to create a performance metric that comprehensively covers the iterations between all benchmarks in a benchmark suite. In the future, this method may allow us to dramatically reduce the number of co-phases that need to be sampled for fine-grain results using the co-phase method. Ultimately, that could allow the co-phase method to scale to much larger numbers of concurrently executing threads.

Bibliography

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded commercial workloads. In *Annual International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[2] J. E. S. Ashutosh S. Dhodapkar. Comparing program phase detection techniques. In *36th International Symposium on Microarchitecture*. ACM, Dec. 2003.

[3] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2005)*, Mar. 2005.

[4] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb. 2003.

[5] 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.

[6] I. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 128–137, Oct. 1996.

[7] T. M. Conte, M. A. Hirsch, and W. W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6):714–720, June 1998.

[8] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, Oct. 1996.

[9] M. Durbhakula, V. S. Pai, and S. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999.

[10] L. Eeckhout and K. De Bosschere. Efficient simulation of trace samples on parallel machines. *Parallel Computing*, 30:317–335, 2004.

[11] L. Eeckhout, S. Eyerman, B. Callens, and K. De Bosschere. Accurately warmed-up trace samples for the evaluation of cache memories. In *Proceedings of the 2003 High Performance Computing Symposium (HPC-2003)*, pages 267–274, Apr. 2003.

[12] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48(4):451–459, 5 2005.

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

[14] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 83–94, Sept. 2002.

[15] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, Mar. 2005.

[16] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO-22)*, pages 236–245, Dec. 1992.

[17] R. M. Fujimoto and W. B. Campbell. Direct execution models of processor behavior and performance. In *Proceedings of the 1987 Winter Simulation Conference*, pages 751–758, Dec. 1987.

[18] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *SIGMETRICS'03*, June 2003.

[19] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[20] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, Sept. 2001.

[21] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2003.

[22] J. Haskins and K. Skadron. Accelerated warmup for sampled microarchitecture simulation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):78–108, Mar. 2005.

[23] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 5(1), 2001.

[24] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[25] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, June 1994.

[26] J. L. Kihm, T. Moseley, and D. A. Connors. A mathematical model for accurately balancing co-phase effects in simulated multithreaded systems. In *Workshop on Modeling, Benchmarking and Simulation (MoBS) held in conjunction with ISCA*, June 2005.

[27] 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*, Sept. 2000.

[28] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, Nov. 1988.

[29] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 57–67, Mar. 2004.

[30] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *Hawaii International Conference on System Sciences*, Jan. 1994.

[31] Y. Luo, L. K. John, and L. Eeckhout. Self-monitored adaptive cache warm-up for microprocessor simulation. In *SBAC-PAD'04*, pages 10–17, Oct. 2004.

[32] D. Marr, F. Binns, D. Hill, G. Hinto, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, 6(1), 2002.

[33] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr. 1965.

[34] S. S. Mukherjee, S. K. Reinhardt, M. L. B. Falsafi, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin wind tunnel ii: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[35] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, June 2005.

[36] A.-T. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M. Michael. Accuracy and speed-up of parallel trace-driven architectural simulation. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, pages 39–44, Apr. 1997.

[37] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC-41*, June 2002.

[38] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, 2000.

[39] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT'03*, pages 244–256, Sept. 2003.

[40] E. Perelman, M. Polito, J.-Y. Bouget, J. Sampson, B. Calder, and T. Sherwood. Detecting phases in parallel applications on shared memory architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Apr. 2006.

[41] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, pages 10–20, Mar. 2005.

[42] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2003.

[43] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *DAC-40*, June 2003.

[44] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2005)*, Mar. 2005.

[45] R. R. Schaller. Moore's law: Past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[46] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, Oct. 1998.

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

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

[49] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, pages 336–349. ACM, June 2003.

[50] R. Shrout. Intel next generation cpu technology - penryn and nehalem. http://www.pcper.com/article.php?aid=382&type=expert, Mar. 2007.

[51] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[52] 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)*, Feb. 2004.

[53] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2003.

[54] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 191–202, 1996.

[55] 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, Nov. 2005.

[56] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 143–153, Mar. 2006.

[57] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Sampling startup for simpoint. *IEEE Mazazine*, 2006.

[58] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Representative multiprogram workloads for multithreaded processor simulation. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Sept. 2007.

[59] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the 2004 IEEE International Symposium on Performanc e Analysis of Systems and Software (ISPASS'04)*, Mar. 2004.

[60] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. A novel evaluation methodology to obtain fair measurements in multithreaded architectures. *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[61] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. Measuring the performance of multithreaded processors. *2007 SPEC Benchmark Workshop*, 2007.

[62] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. In *SIGMETRICS*, June 2005.

[63] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS'96*, pages 68–79, May 1996.

[64] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 79–89, May 1991.

[65] R. E. Wunderlich, T. F. Wenish, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30)*, June 2003.