The Dissertation Committee for Yue Luo Certifies that this is the approved version of the following dissertation:

Improving Sampled Microprocessor Simulation

Committee:

Lizy K. John, Supervisor

Rema Hariharan

Stephen W. Keckler

Earl E. Swartzlander, Jr

Nur A. Touba

Improving Sampled Microprocessor Simulation

by

Yue Luo, B.E.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August, 2005

Dedication

To my wife Junhui

and

my parents.

Acknowledgements

First I would like to thank my advisor, Dr. Lizy John, for her support, advice, guidance and good wishes. Her availability at all times including on weekends, and her dedication to work has had a profound influence not only on my academic pursuit, but also on my life. I still remember that when I was in China (before coming to UT Austin), I sent an email to Lizy and was surprised by the quick reply: it was 4 o'clock in the morning in Texas and she had already started working! I am also grateful for the freedom and the flexibility she gave me throughout my Ph.D. study.

My gratitude goes to the committee members (in alphabetical order), Dr. Rema Hariharan, Dr. Steve Keckler, Dr. Earl Swartzlander, and Dr. Nur Touba, for their invaluable comments, productive suggestions, and the time to read the draft of my thesis.

I would like to thank Dr. Rema Hariharan for giving the opportunity to work at Sun Microsystems. I gained valuable experience developing a new web server benchmark.

I would like to thank the students (past and current) at the Laboratory for Computer Architecture -- Tao, Madhavi, Juan, Ravi, Shiwen, Byeong, Aashish, Ajay, Sean, Lloyd, Jason, Tyler, and Hari. They provided valuable feedback on the drafts of my paper submissions and on my practical talks. I enjoyed working with Ajay and Aashish in writing the research paper.

Dr. Lieven Eeckhout from Ghent University, Belgium provided valuable insight and comments, which helped me improve my research.

Thanks to Debi, Amy, Melanie and other administrative assistants who worked in the Department in the past years.

I would like to thank my parents, my brother, my parents-in-law, and my friends, who have had a tremendous influence on my life.

Last but not least, I am grateful to my wife, Junhui, for her consistent love, support, and encouragement throughout my graduate years. This is not something I could accomplish alone.

Improving Sampled Microprocessor Simulation

Publication No._____________

Yue Luo, Ph.D.

The University of Texas at Austin, 2005

Supervisor: Lizy K. John

Microprocessor evaluation using detailed cycle-accurate simulation is prohibitively time-consuming. Sampling is the most widely used simulation time reduction technique. In this dissertation, new sampling designs that utilize the characteristics of the workload, the microarchitecture being simulated, and the user's specific objective are proposed. They improve accuracy, and reduce simulation time and storage cost.

Statistical sampling theory is employed to study the choice of sampling unit size for simple random sampling with perfect warm-up. More importantly, the inherent characteristic of the benchmarks that affects the choice of sampling unit size is discerned.

Previous research has been focusing on the accuracy of Cycle Per Instruction (CPI). However, most simulations are used to measure the speedup due to some microarchitectural enhancements. A new sampling scheme that employs ratio estimator from statistical theory is proposed to measure speedup and to quantify its error. In the experiment, 9X fewer instructions are simulated as compared to estimating CPI for the same relative error limit.

This dissertation extends sampling techniques to the simulation of commercial workloads such as On-Line Transaction Processing (OLTP) used by banks, airlines, etc. The applicability of simple random sampling and representative sampling for OLTP workloads is investigated. A dynamic stopping rule is proposed for sampling OLTP workloads, which requires only one simulation and thus eliminates the second simulation in previous random sampling methods.

In order to achieve accurate sampling results, microarchitectural structures must be adequately warmed up before each measurement. Previous warm-up techniques have not considered the cache configuration being simulated, an important factor on the warm-up length. This dissertation presents a new cache warm-up technique for sampled microprocessor simulation, which allows the warm-up length to be adaptive to cache configurations and benchmark variability characteristics. As a result, warm-up length has been greatly reduced, especially for small caches, without losing accuracy.

For trace-driven simulation, the sampled traces have to be stored. Another contribution of the dissertation is the Locality Based Trace Compression (LBTC) technique, which employs both spatial locality and temporal locality in program memory references. It efficiently compresses not only the address but also other attributes associated with each memory reference.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

A processor is a complex digital system with tens to hundreds of millions of transistors. It is important to know the performance of a new processor during the design phase. Before a processor is actually built, its performance can be obtained through analytical modeling or simulation. Due to the complexity in modern processors, analytical modeling may not give adequate accuracy. Therefore, simulation of standard benchmarks has been the most important tool for computer architects to study design tradeoffs. However, detailed cycle-accurate simulation is extremely time-consuming. Thus design space exploration through simulation of complete benchmarks is prohibitive. This difficulty arises from two causes.

Firstly, modern benchmarks are no longer small kernels or synthesized toy programs. Instead, they are very close to real-world programs and often take a long time to execute. For example, each program in the SPEC CPU2000 benchmark suite [75] runs for minutes on a real machine. This problem will only become worse in the future. The SPEC CPU committee is gathering programs for the next release, SPEC CPU2005. It is required that the execution time of each program be no less than 10 minutes on a machine with a SPEC CPU2000 baseline metric of approximately 700 for integer codes and approximately 900 for floating point code, which translates into hundreds of billions of dynamic instructions. Database benchmarks usually take even longer to run. TPC-C benchmark [77] is required to run at least 2 hours on a real machine. On today's high-performance machines, it will result in trillions of instructions.

SPEC®, SPECint®, SPECfp®, and SPECweb® are registered trademarks of Standard Performance Evaluation Corporation (http://www.spec.org/). TPC® is a registered trademark of Transaction Processing Performance Council (http://www.tpc.org/).

Secondly, modern superscalar microprocessors are becoming increasingly complex. Following Moore's law, more and more transistors are available on a chip. By taking advantage of the additional transistors, processor architects have designed various microarchitectural enhancements to improve performance or to reduce power consumption. The complexity of processors is inevitably reflected in the simulators that model the processors, and slows down the simulators. In addition, designers and researchers want to simulate not only the processor and the memory subsystem but also the whole computer system. These systems can be very complex, consisting of multiple computers connected by a high-speed network. Each of the machines can have multiple processors and disks. Each processor can be multi-threaded. Although the hardware devices (e.g., multi-processors) usually work in parallel in real world, most modern simulators simulate them sequentially in order to simplify the development of simulators and to maintain the determinism of the simulation result. Thus the more components to be simulated, the slower the simulation will be. The gap between the speed of execution on real machine and the simulation is currently only increasing.

Therefore, despite the faster machines users now have to run the simulations on, the problem of long simulation time is exasperating. As a result, design space exploration by detailed simulation of full benchmarks becomes impractical. SimpleScalar [6] is the most widely used simulator in academic research. Table 1.1 shows the time to simulate selected SPECint2000 benchmarks on a 1GHz Pentium III machine with *sim-outorder*, the detailed out-of-order superscalar simulator from the SimpleScalar 3.0 tool set. It usually takes several days to simulate one program. Some benchmarks in the suite, which could not be fully study in this research, require weeks of simulation time. Assume that a processor designer needs to evaluate 10 candidate configurations. Suppose that the benchmark suite has 26 programs and each program has 3 input sets. If one program-input execution takes 1 week, the total simulation will be 10*26*3=780 weeks, which is about 15 years! If the user has enough processors to run the simulations, then the simulation for each benchmark and each configuration can be run in parallel on different machines. The simulation time will eventually be limited by the simulation of one single benchmark. Therefore, it is very important to reduce the simulation time of each single benchmark program. The problem of long simulation time is much worse in the computer industry than in academic research. Academic simulators often use a simplified model of processors. Simulators in industry for designing real computers, on the other hand, are much more complex and hence much slower. Todi from HP reported that it would take their Itanium simulator 676 days to simulate a benchmark from SPEC CPU2000 that has 146 billion instructions [76].

Table 1.1: Number of instructions and simulation time of selected SPEC INT2000 benchmarks with the reference data set.
The data set name is appended to the benchmark name.

Benchmark	Number of instructions (million)	Simulation time (days)
gcc-166	46,917	2.2
bzip2-source	108,878	4.4
eon-rushmeier	57,870	2.7
gzip-graphic	103,706	7.2
vortex-1	118,976	4.6
vpr-route	84,068	4.1
crafty	191,882	9.3

1.1 TECHNIQUES FOR REDUCING SIMULATION TIME

Several types of techniques have been proposed to reduce the simulation time. One method is to reduce the input data set to the benchmark. The same benchmark program is executed but the problem size to the program is decreased. For instance, suppose that the functionality of a benchmark is compressing files. Then to reduce the execution time, a smaller input file can be used. MinneSPEC, proposed by KleinOsowski and Lilja [35], comprises reduced data sets for SPEC CPU2000. The input sets to the benchmarks are carefully reduced to maintain the function-level execution profile and the instruction mix. Designing a reduced input data set is no easy task: There is no automatic way to do it. It requires good understanding of the source code of the benchmark program. One major disadvantage with this approach is that it is very hard to maintain the characteristics of data accesses. Take matrix multiplication as an example. Reducing the matrix size can easily reduce the instruction count while maintaining the function-level execution profile and the instruction mix. However, the original matrix may be larger than the data cache whereas the reduced matrix may fit in the data cache. There can be much fewer data cache misses for the reduced input data set. Data cache misses have huge impact on the performance of modern processors. Therefore, in this example, the performance evaluation based on the reduced matrices may not give a valid result.

Researchers have also proposed statistical simulation techniques to reduce simulation time [58][56]. In this method, the instruction trace of the benchmark is studied and the characteristics that impact the performance are extracted. These characteristics are a combination of microarchitecture independent ones (e.g., instruction mix and dependence distance between instructions) and microarchitecture dependent ones (e.g., cache miss rate and branch misprediction rate). Then a new and much smaller trace is synthesized based on these characteristics. The new trace is fed to the simulator. Usually the simulation converges very quickly, in thousands of cycles. If microarchitecture dependent characteristics are used, then these aspects of the microarchitecture are not simulated. Instead, an artificial model is used. For example, in HLS [58] caches are not simulated, but cache misses are injected by the simulator based on the cache miss rate from the profiling done before the simulation. Such simplification may affect the authenticity of the simulation result. It is also difficult to determine what characteristics to capture for future microprocessor design. As an example, suppose that the trace were synthesized before value prediction [41] was proposed, then using such a trace to evaluate new processors with value predictors may give a biased result. However, statistical simulations are very useful for early design space exploration because such simulations are very fast and accuracy is not the top concern in early design phases.

As computers are getting cheaper and multi-core processor getting more common, the user can have multiple CPUs to run simulations. Girbal, *et al*. proposed the DiST method, which distributes the simulation of a single benchmark onto multiple computers [24]. Each computer simulates only part of the benchmark. Because the performance of a part depends on the microarchitectural state generated by the previous part, the simulation on different machines have to be overlapped, incurring overhead for DiST. Adding up the simulations on all the machines, the entire benchmark is simulated plus the overhead of overlapped parts. It is better to combine distributed simulations with other simulation time reduction techniques, such as sampling [40].

Sampling is the focus of this dissertation. The next few sections are devoted to detailed discussion of sampled processor simulation.

1.2 SAMPLING

Sampling, the most widely used simulation time reduction technique, is the focus of this dissertation. Sampling has been used in social science and quality control for a long time and it enjoys a solid foundation in statistics. Applying sampling to microprocessor simulation can greatly reduce the simulation time while retaining good accuracy. Sampling can be used for cache simulations and cycle-accurate performance simulations. The focus of this dissertation is on cycle-accurate performance metrics such Cycles Per Instruction (CPI). In sampled simulation[1], the original full instruction stream is divided into *N* non-overlapping chunks of *m* continuous instructions. Each chunk is a basic simulation unit, or a *sampling unit*. The *sampling unit size* is the number of instructions in each chunk. The *population* refers to all the chunks that constitute the entire instruction stream. *Population size* is the total number of sampling units in the entire instruction stream, usually denoted *N* in this dissertation. A *sample* consists of selected chunks that are actually simulated and measured (In practice, more instructions are simulated for warming up microarchitectural structures). The number of sampling units in a sample is the *sample size*, expressed as *n*. The ratio of sample size to the population size is the *sampling fraction*, denoted by the letter *f* (=*n/N*). The CPI of each sampling unit depends not only on the instructions executed in the unit, but also on the initial state of all microarchitectural structures at the beginning of this unit. The initial state is, in turn, the result of the execution of all the instructions before the sampling unit. Executing a limited number of instructions before a sampling unit to get (approximately) correct initial state is known as *warming up* the microarchitecture. The number of instructions used for warm-up before a sampling unit is its *warm-up length*

Figure 1.1 gives the conceptual picture of the instruction stream in a sampled simulation. Only a small portion in the entire instruction stream is measured. A number of instructions before each measured sampling unit are used for warm-up. And the rest of the instructions are "skipped". Whether they can be really skipped depends on the implementation. Figure 1.2 illustrates a simple taxonomy of the implementation of sampled simulations. A sampled simulation can be either *execution-driven* or *trace-*

[1] The terminology in literatures on sampled microprocessor simulation is not consistent. One notable difference is that in some papers a "sample" actually means what is referred to as a "sampling unit" in other papers. Throughout this dissertation, terminology from the traditional statistical sampling theory is used.

driven. In an execution-driven simulation, the system state for the next clock cycle or the next instruction is completely computed from the current system state by the simulator. In an execution-driven simulator, sampling can be done in two ways. In the first method, the simulator starts from an initial state for the whole benchmark and computes every future state from there. During the simulation the simulator alternates between different modes. It does cycle-accurate simulation and measurement for the sampling units that need to be measured. Only the microarchitectural structures requiring warm-up (e.g., caches, branch predictor) are simulated during warm-up. The remaining instructions are only simulated in a fast mode to get the architectural state. In the second method, the initial state for each sampling unit to be measured is stored in a file called the *checkpoint* file. The processor can read a checkpoint file and compute the system state for the sampling unit. If the checkpoint file contains the state of all microarchitectural structures, then no warm-up is necessary. Otherwise, warm-up is still needed. In a trace-driven simulation, the simulator relies on a previously recorded trace file to compute future system state. Usually instruction words, instruction addresses, and data addresses, etc. are stored in a trace file so that there is no need to compute this information from the current system state. Therefore, a mode-switching execution-driven simulation requires minimum disk space because only one initial state (or an executable binary file) for the entire benchmark is stored. For trace-driven simulations, or execution-driven simulations with a checkpoint for each sampling unit, the traces or checkpoints have to be stored, often resulting in large storage cost.

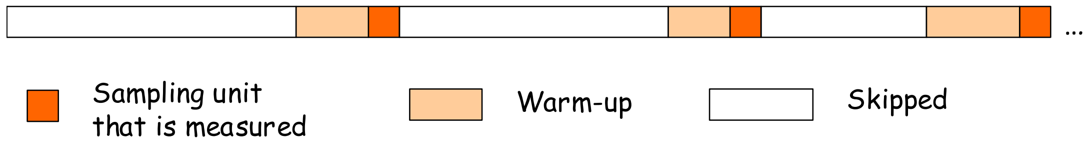

Figure 1.1: Illustration of a sampled instruction stream.

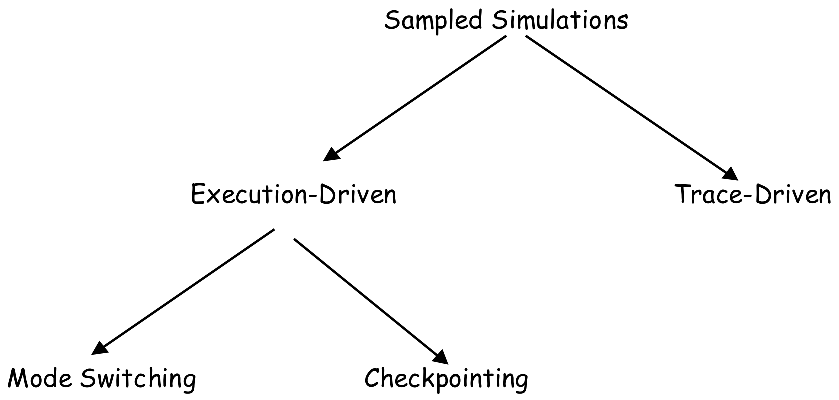

Figure 1.2: Implementation of sampled simulations.

Sampling works well because the execution of benchmarks is usually very repetitive. For example, the size of the binary file of benchmark *bzip2* from SPEC CPU2000 statically compiled for Alpha ISA is only 320KB, but executing it with input set *source* generates about 109 billion instructions. These dynamic instructions are repetitions of the relatively small number of static instructions. In addition, the microprocessor limits the variability in the execution of the benchmark. Suppose that the user wants to measure IPC (Instruction Per Cycle) of the benchmark running on a processor that commits a maximum of 4 instructions in a cycle. The variability in IPC in every cycle is constrained between 0 and 4. As a result, with sampling, simulating only a small number of instructions can give fairly accurate result. Wunderlich, *et al.* showed that under the assumption of no measurement error, CPI can be estimated to within an error of 3% with 99.7% confidence by measuring fewer than 50 million instructions per benchmark [81].

Sampling has several advantages over other simulation time reduction techniques. With most of the techniques the user do not know the error in the simulation result. The only way to find out the error is to run the full simulation and compare with the reduced simulation, which completely defeats the goal of reducing simulation time. Alternatively, the user can rely on the previously published validation of the simulation time reduction technique, but the benchmark and the microprocessor configuration to be simulated in the user's environment are usually different from the published experiment. With a sampling scheme that employs statistical sampling theory, the user can get a confidence interval to quantify the error *without simulating the entire benchmark in detail*. Furthermore, the sample comes directly from the benchmark, so, unlike in the statistical simulation, the user does not need to worry about not capturing some important characteristics in the benchmark for performance evaluation.

1.3 Factors Affecting Sampling Designs

Simulation experiments, especially full-system simulations, are complex. As in any complex project, many factors affect the design of the experiment.

- **Goal of simulation**. In different experiments, users may have different goals. In some experiments, users are happy with just cache miss rates. In others they may want to find out the CPI, or the speedup due to a microarchitectural enhancement, or EPI (Energy Per Cycle), or even the highest temperature the processor will experience during the execution of the benchmark. Different sampling designs may be appropriate for different target metrics.
- **Characteristics of the benchmark**. Different types of benchmarks exhibit different behavior. For example, a commercial benchmark like TPC-C is vastly

different from a CPU-intensive benchmark such as SPEC CPU2000. The differences between benchmark characteristics are not considered in many current sampling methods.

- **Simulation infrastructure**. Simulators are pieces of complex software. Validating a simulator is even more challenging [16]. Once a simulator has been developed and validated, the modification should be kept to a minimum. Therefore, applying a simulation time reduction technique to an existing simulator requires careful consideration. A technique that best fits the simulator should be selected. Different simulators can have different problems. For example, if the user has a trace-driven simulator, then besides reducing the cost of simulation time, the experiment designer also needs to reduce the cost of storing the traces. The computing resource the user has also affects the selection of simulation time reduction techniques. Multiple benchmarks and processor configurations are commonly evaluated in one experiment. If the user has fewer machines than the product of the number of benchmarks and the number of configurations, then distributing the simulation of one single benchmark onto multiple machines may not be important. If, one the other hand, the user has enough computers, then it is desirable to parallelize the simulation as much as possible.

1.4 Problems in Sampled Processor Simulation

The above discussion clearly shows that no single sampling method is the best for every situation. Naturally, the goal of this dissertation is not to find the universally best solution because it simply does not exist. Instead, the objective of this research is to improve sampled microprocessor simulation for different factors, and to let users select the technique according to their particular environment. Specifically, the following problems are attacked.

- There is no consensus in previous research on how to choose a good sampling unit size. Given a fixed simulation time budget, what sampling unit size should the user choose? What inherent characteristic of the benchmarks, if any, should affect the user's choice of sampling unit size?
- Nearly all previous research on sampling focuses on CPI but in many experiments users want to find out the performance impact of a microarchitectural enhancement. They are more interested in the speedup than in the absolute value of CPI. How does measuring speedup affect design of sampling experiment? Is there a way to further reduce the simulation time but maintain the accuracy?
- The cache warm-up process is affected not only by the benchmark but also by the cache configuration being simulated. However, previous cache warm-up methods only consider the characteristics of the benchmark. How can a better cache warm-up scheme be designed that adapts to the cache configuration?
- Most of the sampling methods are designed and validated for CPU intensive benchmarks such as SPEC CPU2000. Commercial benchmarks such as On-Line Transaction Processing (OLTP) workloads are significantly different. Are those sampling methods applicable to commercial benchmarks? Can better sampling methods be designed for commercial benchmarks?
- In trace-driven simulations, the sampled instruction traces have to be stored. Trace files, especially those with extended information for each instruction, can be huge. How can those trace files be better compressed?

1.5 THESIS STATEMENT

Detailed simulation of microprocessors is prohibitively time-consuming. Sampling designs that utilize the characteristics of the workload, the microarchitecture being simulated, and the user's goal for simulation can reduce the simulation time and storage cost with very little loss of accuracy.

1.6 CONTRIBUTIONS

This research makes multiple contributions to sampled processor simulation.

Utilizing the intracluster correlation coefficient from statistical sampling theory, this study finds that using large sampling units is not as effective as using small sampling units at improving the accuracy given the same simulation budget. It also provides insight into the inherent characteristic of the benchmarks that favors small sampling unit sizes.

The applicability of two sampling techniques, representative sampling and simple random sampling, is studied for OLTP workloads. The chunk size is found to be an important parameter in representative sampling. To successfully apply representative sampling, the user needs to carefully choose the chunk size. A dynamic stopping rule for simple random sampling is proposed. It eliminates the second round of simulation often required in the previous techniques, thus it improves usability and reduces simulation time.

By employing the ratio estimator from statistical sampling theory, an efficient sampling method is designed to measure speedup and to quantify its error. It is shown that to achieve a given relative error limit for speedup, it is not necessary to estimate CPI to the same accuracy. In the experiment, estimating speedup requires about 9X fewer instructions to be simulated in detail in comparison to estimating CPI for the same relative error limit. Therefore using the ratio estimator to evaluate speedup is very cost-effective and offers great potential for reducing simulation time.

A new technique for warming up microprocessor caches is proposed. The simulator monitors the warm-up process of the caches and decides when the caches are warmed up based on simple heuristics. In the experiments the proposed Self-Monitored Adaptive (SMA) warm-up technique on average exhibits only 0.2% warm-up error in CPI. SMA achieves smaller average warm-up error with only 1/2~1/3 of the warm-up length of previous methods. In addition, it is adaptive to the cache configuration simulated. For simulating small caches, the SMA technique can reduce the warm-up overhead by an order of magnitude compared to previous techniques. Finally, SMA gives the user an indicator of warm-up error at the end of the cycle-accurate simulation that helps the user to gauge the accuracy of the warm-up.

To reduce the storage cost for sampled trace driven simulation, a new trace compression method, Locality Based Trace Compression (LBTC), is proposed. It employs both spatial locality and temporal locality in program memory references. It efficiently compresses not only the address but also other attributes associated with each memory reference. It gives better compression ratio than previous methods. In addition, LBTC is designed to be simple and on the fly.

1.7 ORGANIZATION

Chapter 2 surveys previously proposed microarchitectural sampling and warm-up techniques for processor simulation.

Chapter 3 deals with the problem of selecting good sampling unit sizes. Statistical sampling theory is employed to tackle this problem. More importantly, the inherent characteristic of the benchmarks that affects choice of sampling unit size is identified.

Chapter 4 studies the applicability of simple random sampling and representative sampling to OLTP workloads. A new dynamic stopping rule for simple random sampling is proposed and evaluated.

Chapter 5 presents a more efficient sampling method for measuring the speedup for microarchitectural enhancements. The proposed method is experimentally evaluated and the reason for its improved efficiency is investigated.

Chapter 6 discusses the problem of warm-up and reviews previous warm-up techniques. Then a new self-monitored adaptive technique for cache warm-up, which overcomes a major weakness of previous methods, is proposed and evaluated.

Chapter 7 proposes Locality-Based Trace Compression (LBTC) technique. It is compared with previous techniques and is shown to be more effective at compressing trace files with extended information.

Chapter 8 concludes the dissertation by summarizing the contributions and suggesting future opportunities.

Chapter 2. Previous Research

This chapter briefly surveys previous research on sampling techniques for processor simulation. Simulation can be used to measure different performance metrics, such as CPI, EPI (Energy Per Instruction), and cache miss rates. Most early techniques are designed for cache simulation to measure cache miss rate. More recent research primarily deals with cycle-accurate simulation for CPI or EPI. This dissertation focuses on cycle-accurate simulation and cache warm-up. Therefore, this chapter is divided into to two sections on the two topics respectively. Caches are usually the most difficult-to-warm-up microarchitectural structures in a processor, so most of the research on sampled cache simulation is actually about warm-up methods and thus they are surveyed with warm-up techniques in Section 2.1. Cycle-accurate simulations are reviewed in Section 2.2. However, many papers on sampling techniques encompass both topics on sampling per se and on warm-up issues. They are described in one of the two sections depending on which topic is the focus in that paper. A paper may also appear in both sections if needed.

2.1 Cache Simulation and Cache Warm-up

The problem of cache warm-up is that the state of the cache is unknown at the beginning of each sampling unit. In other words, since portions of the trace are unexamined between observations, it is unknown whether the first reference to each cache block will be a hit or a miss. Such references are referred to as *cold-start references*. If all cold-start references are assumed to result in cache misses, it is called the *cold* scheme, which is equivalent to assuming all cache lines to be initially invalid for every sampling unit. Laha, *et al.* employed this method for small caches [37]. They reasoned that small caches would be purged upon a context switch so they select the sampling units starting at a context switch. Large caches, however, may not be completely flushed at a context switch. Some information is always retained in caches larger than 16KB across a context switch. Therefore, they proposed not counting these cold-start references when calculating cache misses for large caches. This effectively assumes that the miss rate for the cold-start references is equivalent to the miss rate for all other references. In their experiment, sampling unit size of 5,000, 10,000, and 20,000 instructions were used. It was shown that cache miss per instruction (MPI) can be accurately estimated with a sample size of 35.

Wood, *et al.*, however, showed that Laha, *et al.*'s assumption about large caches is usually not true [80]. The miss rate for the cold-start references is higher than the overall miss rate. Employing a renewal theoretical model, they proposed a method called INITMR to estimate the miss rate for the cold-start references by observing the average live and dead time for each cache line. Kessler, *et al.* evaluated INITMR against other warm-up methods [34]. INITMR can be used to calculate the cache miss rate from sampled trace, but not directly applicable to microarchitectural simulation to get CPI. Therefore, it is not further discussed here.

Fu and Patel also realized that the cold-start references show a miss rate higher than the overall miss rate [23]. They divided each sampling unit into a priming interval and a evaluation interval. Cache miss rate is only measured in the evaluation interval, not in the priming interval. The priming interval is initially simulated to warm up the cache. This method is called the *prime* scheme. The *prime-xx%* method refers to devoting *xx%* instructions from the sampling unit to warm-up. The *prime*-50% scheme is also called *half* in the literature [13]. During the priming interval, *miss-distance* is recorded, which is the number of references between misses including the first miss. In the evaluation interval, the following steps are used to predict whether each cold-start reference is a hit or a miss based on the miss-distance history and the cache contents. First, in the priming interval if a miss occurs, then the miss distance is calculated and stored in a small history table, which is a list of the most recent miss distances. Next in the evaluation interval, upon a cold-start reference, the miss distance *d* is calculated. Prediction is made according to the following criteria:

- If the history table is empty (i.e. no misses have been recorded), then predict a hit.
- Else if *d* is within the range of distances recorded in the history table, then predict a miss.
- Else if a prediction cannot be made based on the history, the contents of the cache are searched. If the adjacent sets hold addresses of the adjacent memory blocks to the memory block being loaded, a hit is predicted, else a miss is predicted.
- Else if none of the above conditions are met, predict a miss.

The initial cache state for a sampling unit can be also assumed to be the same as the state at the end of the last sampling unit. Warm-up techniques employing this assumption, such as those proposed by Agarwal, *et al.* [1], are called *stitch*. It is like stitching all the sampling units together to create a large continuous chunk of instructions. The accuracy of the stitch scheme depends on how much of the cache state has been replaced between two sampling units and how much of the changed state is accessed during the second sampling unit. If most of the cache blocks are flushed as after a context switch, then the accuracy will be impaired. Crowley and Baer [13] compared different sampling techniques for cache simulation in the context of 5 Windows NT desktop applications (Adobe Acrobat Reader, Netscape Navigator browser, Adobe Photoshop, Microsoft PowerPoint, and Microsoft Word). They compared cold, INITMR, prime-20, half, stitch and some varieties of these techniques. They concluded that for the determination of cache miss ratios, stitch and INITMR are the best at overcoming the difficulties inherent with the problem of the cold-start references at the beginning of each sampling unit. Using these sampling techniques resulted in the accurate determination of cache miss ratios for caches of sizes up to 64KB.

For single-level, write-through, write-allocate, LRU replacement caches, there exist cache simulation algorithms that can simulate multiple cache configurations in a single run. Conte, *et al*. combine such algorithms with sampled simulation [11]. They assume that the entire instruction stream is available although cache miss rates are only measured during selected sampling units. By continuous recording some information throughout the simulation of the entire instruction stream, the cache can be kept warm between sampling units using an LRU stack. Thus, the warm-up error is minimum for the cache simulation.

Of course, for sampled cycle-accurate simulation, the most accurate way to warm up the caches is to do cache simulation throughout the benchmark execution. This is how Jimeno-Ochoa, *et al*. did in their Warm Time Sampling scheme [31]. But this work was largely unnoticed in the computer research community. On ISCA 2003, Wunderlich, *et al*. proposed a similar approach, SMARTS (Sampling Microarchitecture Simulation) [81]. The simulator switches between functional warm-up and cycle-accurate simulation. During the functional warm-up, the simulator executes the program without simulating the pipeline stages, but the caches and the branch predictors are simulated. During the cycle-accurate simulation, the simulator models every microarchitectural structure cycle by cycle. Therefore, the only error in warm-up is introduced by not simulating the effect of out-of-order execution and wrong path execution on the caches during functional warm-up. It has been shown that this error is small [81][8]. Although this warm-up scheme is by far the most accurate, it is still not satisfactory. First, always simulating caches can be a waste of resource. According to sampling theory, for a specific accuracy, the sample size should be determined by the variability in the population. If the benchmark does a lot of repetition, only a tiny fraction of the instruction stream is needed. However, the scheme requires that caches be simulated for every instruction, which is inefficient. Secondly, always warming up the cache makes distributed simulation hard. For sampling methods such as SimPoint [69] and Variance SimPoint [61], where a small number of relatively large sampling units are taken, each sampling unit can be simulated in parallel on different machines to greatly improve the overall simulation speed. However, constantly warming up caches makes it difficult to distribute the simulation on multiple machines.

Nguyen, *et al*. proposed the following equation to calculate the warm-up length [55].

$$W = \frac{C / L}{m * r},$$

where C is the cache size in bytes, L is the cache line size in bytes, m is the cache miss ratio and r is the number of memory references per instruction. They also proposed distributing sampling units on multiple machines in parallel to speed up the simulation. The problem with this approach is that the cache miss ratio to calculate the warm-up length is unknown before simulation. Actually, it is exactly what the user is trying to estimate through sampled simulation.

Haskins and Skadron proposed the Minimal Subset Evaluation (MSE) technique [28], which uses formulas derived from combinatorics and probability theory to calculate, for some user-chosen probability p, the number of memory references prior to each sampling unit that must be modeled in order to achieve accurate cache state. This work is applicable to only one level of cache but most modern processors employ a hierarchy of caches.

The two most recently proposed state-of-the-art cache warm-up methods are MRRL [29], also by Haskins and Skadron, and BLRL [18][19] by Eeckhout, *et al.* Both methods rely on the same premise on a cache with LRU replacement: For a single level LRU cache, if a memory address is referenced, one knows whether the next reference to the same address results in a hit or a miss. Let $R(a, n)$ denote the nth memory reference to address a, and let $I(a, n)$ denote the dynamic instruction that generates the memory reference. Suppose that the we want to know whether $R(a, n)$ incurs a cache miss or a cache hit. Then we need to find out whether $R(a, n\text{-}1)$ has been removed from the cache. With LRU replacement, a cache line can only be replaced by a newer memory reference. Thus by examining all the memory references between $R(a, n\text{-}1)$ and $R(a, n)$ we will know the result for $R(a, n)$ and there is no need to look further back. Therefore, simulating from $I(a, n\text{-}1)$ will tell us the result for $R(a, n)$. This premise is no longer true for the level 2 cache when the level 1 caches employ write-back policy, but experiments show that MRRL and BLRL still work well for multilevel cache simulations.

Based on the above premise, Haskins and Skadron [29] employ the concept of Memory Reference Reuse Latency (MRRL), which refers to the number of dynamic instructions between $I(a, n\text{-}1)$ and $I(a, n)$. The *pre-sample* of a sampling unit refers to the instructions before this sampling unit up to the end of the previous sampling unit. Instructions in a sampling unit and its pre-sample are profiled to get the empirical distribution of MRRL. Given a p-value ($p\%$) the warm-up length is the p-percentile of the distribution. Figure 2.1 gives an example of the empirical Cumulative Distribution Function (CDF) for MRRL. Because MRRL is grouped into bins during profiling the CDF is rugged and exhibits small steps. In the example, warm-up length for p-value of 90% is shown. The MRRL technique suffers from the fact that the distribution of MRRL may change in the instruction stream. For example, the distribution at the beginning of the pre-sample may be different from that of the sampling unit. The CDF from profiling is only an averaged distribution and may not be optimal for the sampling unit. Considering that most of the instructions used to calculate the distribution of MRRL come from the pre-sample, it is hard to guarantee that the instructions in the sampling unit follow the same distribution.

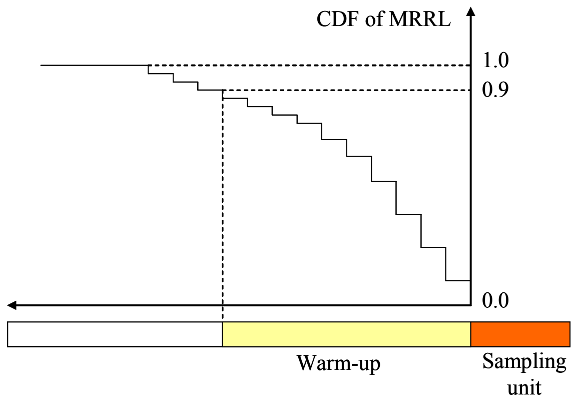

Figure 2.1: The MRRL cache warm-up scheme.

To avoid this problem, Eeckhout, *et al.* proposed the Boundary Line Reuse Latency (BLRL) method [18][19], in which every memory reference in a sampling unit is directly examined instead of relying on the aggregated distribution. Suppose *I*(*a*, *n*) is the first instruction in the sampling unit that references memory address *a*. The instructions in the pre-sample are scanned backward during profiling to search for *I*(*a*, *n*-1). According to the above premise, warming up from *I*(*a*, *n*-1) can guarantee that we know whether *I*(*a*, *n*) incurs a cache hit or a cache miss. Given a *p*-value like 80%, the warm-up length for the sampling unit is chosen such that 80% of the unique references in the sampling unit whose addresses are referenced in the pre-sample are covered by the warm-up instructions. An example adapted from the BLRL paper [18] is given in Figure 2.2. There are 5 unique memory references in the sampling unit whose addresses are also found in the pre-sample, namely, *a*, *b*, *c*, *d*, and *e*. If we start warm-up from the 3rd *b* in the pre-sample, we will cover 80% of the 5 memory references (i.e. *a*, *b*, *c*, and *d*). The only memory references in the sampling unit whose result is unknown are *e* and *g*.

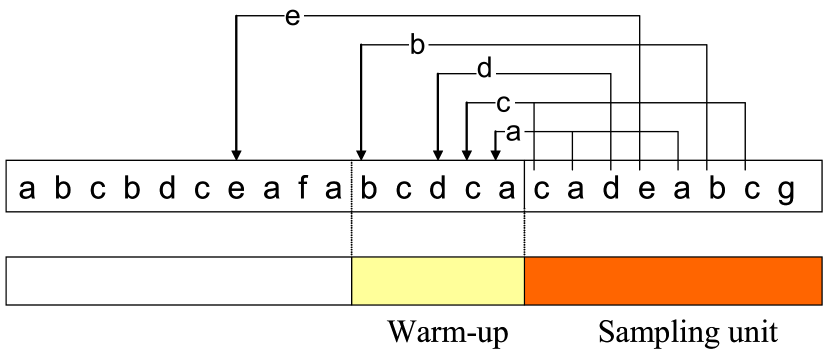

Figure 2.2: The BLRL cache warm-up scheme.

Set sampling is another type of sampling techniques for cache simulation. There is no known method to apply set sampling to cache warm-up for cycle-accurate simulation. But for completeness, set sampling is briefly mentioned here. All the techniques discussed hitherto are often referred to as *time sampling* because the sampling is done in the time dimension (i.e. the instruction stream is sampled). In *set sampling*, however, the sets in the cache are sampled rather than the instruction stream. The sets can be sampled randomly or based on the information about the parameters of the cache. Liu and Peir proposed a two-step set sampling [43]. In the first step, a partial run of the benchmark is simulated with the whole cache to obtain the information about the behavior of each set in the cache. Based on this information, certain sets are selected for inclusion in the sample. In the second phase, the whole benchmark is simulated but only on the selected cache sets, from which the overall cache miss rate is estimated. Kessler, *et al*. proposed a set sampling method called the constant-bits method, which can simulate a hierarchy of multi-megabyte caches [34].

2.2 Sampling for Processor Simulation

Skadron, *et al.* identified a chunk of 50 million instructions from each SPEC INT92 benchmarks to represent the benchmark [71]. To accurately choose the simulation window, they measured the interval branch misprediction rate for each of the benchmarks: i.e. the misprediction rate computed separately over each million-instruction interval in the program. This exposed representative segments of the trace. They also obtained interval traces for data- and instruction-cache miss rates and ensured that the chosen simulation window was suitable with respect to these data as well. They observed that many programs had an initial phase, which was very different from the rest of the execution. The initial phase should be avoided when selecting the chunk of instructions for reduced simulation.

When modeling the performance of the PowerPC 603 processor, Poursepanj simulated 200 sampling units from each SPEC INT92 benchmarks [64]. Each sampling unit consisted of 5,000 instructions. The geometric mean of the IPC for the sampled traces of the SPEC INT92 benchmark suite was within 2% of the true value. However, the error margin for an individual benchmark could go up to 13%.

Lauterbach employed an iterative sampling-verification-resampling method in his study [40]. An initial sample of 100 units of 100,000 instructions each was used. The sampling units were taken at random instruction intervals in the execution of the benchmark. In the verification step, the instruction frequency, basic block density and cache statistics of the sampled traces were checked against the full trace for the benchmark. These metrics could be obtained faster than IPC. In cases where the sample trace was not representative of the full trace, additional sampling units were collected until the required criterion was reached. Final validation was done by simulating several microarchitectures using the sampled trace and comparing the result to the simulation of the full trace. They showed that the absolute performance of samples was within 2% of the performance results of the complete trace.

Iyengar, *et al*. proposed a new metric, called the R-metric, to evaluate the representativeness of the reduced traces when applied to a wide class of processor designs [30]. A basic block annotated with the history of its preceding branch is referred to as a qualified basic block. A basic block that is qualified by the branching history of length *k* and by the preceding *n*-1 qualified basic blocks is called a fully-qualified basic block with parameters *n* and *k*. The R-metric measures the deviation in the reduced trace from the expected scaled count for each fully-qualified basic block. This deviation is expressed as the ratio of instructions that have an incorrect environment in the reduced trace. An ideally representative reduced trace will have a R-metric value of 0. They also proposed a graph-based heuristic to generate reduced traces based on the notions incorporated in the metric. They sampled from the original trace at the granularity of one basic block to minimize R-metric and maximize the representativeness of the reduced trace. Their method was designed for processor models with infinite cache. Therefore, the method does not consider the impact of cache misses and thus will not give very accurate results for simulating real processors with caches.

Sherwood, *et al*. proposed a methodology called Basic Block Distribution Analysis to find a single simulation point in benchmarks [68]. A basic block is a sequence of instructions in a program with a single entry point, single exit point, and no internal branches. A Basic Block Vector (BBV) is a vector of length equal to the number of static basic blocks in the code. Each interval (a chunk of 100 million dynamic instructions in sequence) is characterized by a BBV with each element of the vector showing the frequency of occurrence of a particular static basic block. A BBV is derived for the whole program, called the target BBV, and each entry in the BBV is normalized to total basic blocks, so that sum of all the entries in a BBV is one. Similarly, BBVs are derived for each interval of 100 million instructions and then compared with the target BBV. The comparison is directly made by subtracting one BBV from the other and adding up the absolute values of the difference of each element. The number lies between 0 and 2. The difference of 0 indicates perfect match and 2 indicates a perfect miss-match. A single simulation point is selected by finding the interval with the lowest difference.

Liu and Huang observed that computer programs rely heavily on repetition to perform any significant operations, and that repeated execution of the same code could yield very similar behavior [42]. Based on these observations, they proposed a 3-step sampling scheme called EXPERT (Expedited simulation eXploiting Program bEhavior RepeTition):

1. Partitioning: divide an application into static code sections,
2. Characterization: characterize the behavior repetition of these sections, and
3. Selective simulation: use the characterization to control the degree of sampling in an architectural simulation.

They show that for a set of 22 SPEC CPU2000 applications, the simulation time can be reduced to a few hours or even several minutes if checkpointing is used.

Much research work in sampled simulation follows an ad-hoc approach: the newly proposed technique is evaluated solely experimentally in a few test cases to demonstrate its accuracy. Conte, *et al.* were one of the first to apply statistical theory to processor simulation [12]. The statistical sampling approach allows a confidence interval to be calculated to quantify the accuracy of the simulation *without simulating the whole instruction stream.* The authors also showed how to determine the sample size based on the target accuracy.

The SMARTS method, whose warm-up technique has been discussed in the previous section, also employs sampling theory to calculate the confidence interval and to select the sample size at a given accuracy requirement. Systematic sampling is used in SMARTS but it has been found to be equivalent to simple random sampling. The sampling unit size is 1,000 instructions. The SMARTS method usually involves two simulations. Before the simulation, the user sets a target accuracy expressed as a relative error limit at a certain confidence level. In the first simulation, the user chooses a sample size based on previous experience or an educated guess. After the simulation, the confidence interval for CPI can be calculated. In the lucky but rare case in which the confidence interval is equal to the target accuracy, the second simulation is not needed. If the initial sample size is too large, then the confidence interval will be much narrower than the target accuracy. The second simulation is not needed, either. But the performed simulation is overkill and the user has already wasted time on simulating some unnecessary sampling units. If the initial sample size is too small, then a second simulation must be done. With the result from the first simulation, the sample size for the second simulation can be fairly accurately calculated. The result of the second simulation is expected to just meet the target accuracy. It was shown that CPI could be estimated to within an error of 3% with 99.7% confidence by measuring fewer than 50 million instructions per benchmark for SPEC CPU2000.

Recently, sampling techniques that take advantage of the phase behavior in the programs have been proposed. I call this type of techniques *phase based representative sampling*, or simply *representative sampling*. A phase can be defined as a portion of dynamic execution of a program in which most of the performance metrics such as CPI, show very little variance. In this definition, parts of a program that are disjoint in time may belong to the same phase as long as they show similar values for performance metrics. Since the performance metrics remain stable in a phase, simulating only one chunk of instructions in the phase can give fairly accurate estimation of the performance for the entire phase. If one chunk of instructions from every phase is selectively simulated, the simulation time can be greatly reduced with little loss of simulation information in the whole program.

SimPoint, proposed by Sherwood, *et al.*, is the most acknowledged representative sampling technique [69]. SimPoint also uses BBV for phase classification. BBV for every 100-million-instruction chunk is collected. BBV is usually high dimensional (thousands to hundreds of thousands), and hence random projection [14] is performed on the data to reduce the dimensionality to 15 before using k-means clustering to form interval clusters with similar BBVs. Each cluster corresponds to a phase in the program execution. The clustering algorithm forms clusters for different number of clusters (*k*) and picks the best solution, determined by BIC (Bayes Information Criterion) [33][60]. The simulation point that is closest to the centroid of a cluster is selected as the cluster representative. The cluster representatives (intervals) together form the simulation points of the programs. After selecting the simulation points, the CPI of the whole program can be calculated as a weighted average of CPI values from each of the representative intervals weighted by the cluster size.

Early SimPoint and Variance SimPoint, proposed by Perelman, *et al.*, are two extensions to SimPoint. The chunk sizes are reduced to 1 million and 10 million instructions. Early SimPoint tries to find simulation points early in the program's execution without compromising the accuracy. It reduces the time required for fast-forwarding where check-pointing is not possible. Variance SimPoint uses statistical analysis to guide the choice of number of clusters for a user specified confidence interval and probabilistic error bound for CPI. The confidence interval is valid only on the microarchitecture for which the user does verification, not on the microarchitecture that the user actually uses Variance SimPoint. Nonetheless, Variance improved accuracy over the original SimPoint.

Todi proposed SPEClite, another representative sampling method for SPEC CPU2000 benchmarks [76]. The approach consists of collecting performance metrics using the performance monitoring counters for every interval of 1 million instructions and then using clustering to find representative intervals for phases. The main drawback of this technique is that since the measured phase classification features are for a particular machine, the clusters may not be valid for other microarchitecture configurations.

The representative sampling method proposed by Srinivasan, *et al.* employs χ^2–test instead of clustering algorithms to identify phases [73]. They defined a Chi-square-based Similarity Measure (CSM) to measure the similarity between instruction chunks. CSM compares the sampled IPC distribution to the original IPC distribution to efficiently detect phase changes. Although CSM is microarchitecture-dependent, it was shown that the result is generally accurate on similar microarchitectures.

The sampling method developed by Lafage and Seznec is for cache simulation, not for cycle-accurate processor simulation [36]. But because it is also a representative sampling technique, it is briefly discussed here. This method selects representative slices of program execution based on a microarchitecture-independent feature, reuse distance expressed in terms of instructions executed between two accesses to the same address. They used hierarchical clustering to classify program slices of 1 million instructions. Their results showed an average relative error of 1.52% in data cache miss-rate for the SPEC CPU95 suite.

All of the above research focuses on SPEC CPU benchmark suite. Despite the importance of commercial workloads in the real business world, their simulation methodology has not been thoroughly studied. Patil, *et al*. applied SimPoint methodology to commercial workloads running on Intel Itanium machines [59]. A method similar to SimPoint was used. The code was instrumented with PIN, a tool for dynamic, user-defined instrumentation of Itanium/Linux Programs [62]. The instrumented program was run on a real Itanium machine to collect the BBV profile for every 250-million-instruction chunk. Then clustering analysis from SimPoint is performed to select representative simulation points. Their result showed that representative sampling worked well for their benchmarks. However, most of their benchmarks were run in single-threaded mode. They had one set of multi-threaded programs, SPECOMP2001. They noted the difficulty in studying SPECOMP2001 in their experiment due to the non-determinism and the non-repeatability of multi-threaded programs on real machines. Their experiment on multi-threaded commercial workload was inconclusive.

Chapter 3. On Sampling Unit Size

3.1 Introduction

In sampled simulation of microprocessors, one basic problem is to determine the best sampling unit size. For example, suppose users have a budget of simulating 500 million instructions. To achieve a small error in CPI or IPC, should they simulate 1 chunk of 500 million instructions each or 500 chunks of 1 million instructions each? And why? Despite its importance, there is no consensus on the problem in previous research. Researchers have proposed various sampling unit sizes ranging from 1,000 instructions to hundreds of millions of instructions. In practice, only one large chunk of consecutive instructions are often simulated. I call it *one-chunk sampling*, a special case of sampling with sample size of one. The single sampling unit is becoming larger and larger. Consider the papers in MICRO 2001 and 2003 that used one-chunk sampling to simulate SPECcpu2000. In MICRO 2001, the sampling units used in the six papers were 200 million instructions (in 3 papers), 300 million instructions (in 2 papers), and 10-25 million instructions (in 1 paper). In MCIRO 2003, the sampling units became much larger in 7 papers: 100 million instructions (in 2 papers), 500 million instructions (in 3 papers), 1 billion instructions (in 1 paper) and 5 billion instructions (in 1 paper). Intuitively, simulating more instructions will give a more accurate result, but is this an effective way to improve simulation accuracy?

The final error of sampling simulation comes from two sources. The first source is the inaccuracy in measuring the CPI of each sampling unit, which comes mainly from the *warm-up error*. Because only limited number instructions before each sampling unit are simulated, the initial microarchitectural state at the beginning of a sampling unit is only approximately correct. The final error in CPI caused by approximation in warm-up is called warm-up error. The second type of error, which is the focus of this chapter, comes from sampling itself. Because only part of the instruction stream is simulated, the true CPI for the part that is not simulated is unknown and can only be estimated. Therefore, the estimation of the CPI for the whole benchmark will always have inaccuracy in it. This *sampling error* is inherent to all sample designs.

Warm-up error can be very small if caches and branch predictors are functionally simulated throughout the benchmark execution [81][31]. In this chapter, the warm-up error is assumed to be zero and the warm-up overhead is assumed to be constant. In actual simulation the warm-up overhead depends on the specific warm-up method, so this assumption allows the decoupling of the warm-up issue from the benchmark and the microarchitecture configuration. The assumption of ideal warm-up enables this study to focus on the inherent property of the benchmark instruction stream instead of being tied down to a particular warm-up scheme.

One advantage of sampling over other simulation time reduction techniques is that sampling enjoys a solid mathematical foundation. In this chapter, statistical sampling theory is employed to study the problem of sampling unit size. This study tries to determine how large the sampling unit should be in order to achieve certain simulation accuracy while simulating as few instructions as possible. The intracluster correlation in programs is used to evaluate the effectiveness of large sampling units. It is found that most benchmarks show positive intracluster correlation, which favors a small sampling unit. The inherent property of the benchmarks that causes the positive correlation is also investigated.

This chapter is structured as follows. Section 3.2 presents the statistical theory employed in the study. The experiment and the analysis of the results are shown in Section 3.3. In Section 3.4, the underlying reason for the observation is discussed, which shows that the observation is not a coincidence but caused by an inherent property of many benchmarks. Section 3.5 summarizes the chapter.

3.2 Statistical Sampling Theory

Simple random sampling is assumed in this chapter. In practice, systematic sampling is often used for convenience but it is shown to be equivalent to simple random sampling in processor microarchitectural simulation [81]. In a sampled simulation, the CPI of each sampled unit is measured (y_i, i=1, .., n). The CPI of the full simulation (population mean, $\bar{Y}$) is estimated as2

$$\hat{\bar{Y}} = \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{3.1}$$

That is, the sample mean ($\bar{y}$) is used as an estimator for the population mean, which is intuitive.

A confidence interval can be used to quantify the error of the sampling result. When the sample size is large, the sample mean approximately follows normal distribution, the confidence interval for the population mean at confidence level (1-α) is

$$(\bar{y} - z_{1-\alpha/2}\, S_{\bar{y}},\ \bar{y} + z_{1-\alpha/2}\, S_{\bar{y}}) \tag{3.2}$$

where $z_{1-\alpha/2}$ is the (1-α/2) quantile of a unit normal distribution, and $S_{\bar{y}}$ is the standard deviation or standard error of the sample mean. $V(\bar{y}) = S_{\bar{y}}^2$ is the variance of the sample mean. Because the variance and standard deviation are directly related to the confidence interval, they are used in sampling theory as the indicator of the sampling error and are used to evaluate sample designs. Therefore, the variance or the standard deviation of the sample mean is used to compare the accuracy for different sampling unit sizes hereafter.

For simple random sampling, when the sampling fraction is small,

2 Capital letters refer to characteristics of the population and lowercase letters to those of the sample. The symbol ^ denotes an estimate of a population characteristic made from a sample.

$$V(\bar{y}) = \frac{S^2}{n} \cdot \frac{(N-n)}{N} = \frac{S^2}{n}(1-f) \doteq \frac{S^2}{n} \tag{3.3}$$

where S^2 is the variance of the population:

$$S^2 = \frac{1}{N-1}\sum_{i=1}^{N}(y_i - \bar{Y})^2 \tag{3.4}$$

According to Equation 3.3, as more sampling units are measured (i.e. the sample size n is increased), the variance decreases with $1/n$.

There are two methods to improve the accuracy of the sampling result. In the first method, larger sampling units are used. M consecutive sampling units are grouped together to form a large sampling unit, which is called a *cluster* in the sampling theory. Assume that there are N clusters in the population and n clusters are randomly taken from it. Let y_{ij} be the observed value for the jth unit within the ith cluster. $\bar{\bar{Y}}$ denotes the mean per (small) sampling unit for the population. In the second method, the original sampling unit size is kept, but the sample size is increased to $M*n$ by taking more sampling units.

By applying the two methods to processor simulation, the user still measures the same number of instructions in the cycle-accurate mode. The difference is that, in the first case, n large chunks of instructions are simulated, whereas, in the second case, $M*n$ small chunks of instructions are simulated. So the question is: is using larger sampling units more effective at improving the accuracy?

Calculate the ratio of the standard deviations of the two cases can answer the question. It can be proved that

$$\frac{S_{large}}{S_{small}} \doteq 1 + (M-1)\rho \tag{3.5}$$

where

$$\rho = \frac{\sum_{i=1}^{N}\sum_{j \neq k}^{M}(y_{ij} - \bar{\bar{Y}})(y_{ik} - \bar{\bar{Y}})}{(M-1)(NM-1)S^2} \tag{3.6}$$

The numerator of Equation 3.6 shows that ρ is the average of the correlation between any two sampling units within a cluster. Hence it is called the *intracluster correlation coefficient*. The possible range for ρ is $-\frac{1}{M} \le \rho \le 1$. When ρ=0, the sampling units within a cluster are independent of each other and S_{large}=S_{small}, which means that using a larger sampling unit is as effective as taking more smaller sampling units. When ρ>0, the sampling units within a cluster are positively correlated and S_{large}>S_{small}, so using a larger sampling unit is less accurate. Please note that in Equation 3.5 ρ is multiplied by (M-1). Even a small positive correlation may cause noticeable difference between S_1 and S_2. When ρ<0, using a larger sampling unit is more effective at improving accuracy (S_{large}<S_{small}). In the extreme case ($\rho = -\frac{1}{M-1}$), the large sampling unit reduces the sampling error to zero.

3.3 Experiments and Results

The theory in the above section shows that the effectiveness of a large sampling unit depends on the intracluster correlation coefficient. Experiments are conducted to measure this instracluster correlation. The processor configuration is shown in Table 3.1. The same configuration has been used in study on warm-up [29] and in validation of SimPoint [22].

Although possible sampling unit sizes range from hundreds to billions of instructions, two ranges are studied in the experiments. 1 million to 100 million instructions is the "promising range". It is good candidate for designing new sampled simulation techniques for the future. Good warm-up methods can be devised for sampling unit sizes in this range [29][18][19][48]. The other range of sampling unit size is from 100 million to several billion instructions. This is the unit size for one-chunk sampling commonly used in practice. Baseline sampling unit sizes of 1 million instructions and 100 million instructions are appropriate for studying the two ranges respectively.

Table 3.1: Processor configuration.

Pipeline	
Issue Width	8 instructions/cycle
Decode Width	8 instructions/cycle
Register Update Unit	128 entries
Load-Store Queue	32 entries
Commit Width	8 instructions/cycle
Cache Hierarchy	
L1 Data	16KB; 4-way assoc., 32B lines 2-cycle hit
L1 Instruction	8KB; 2-way assoc., 32B lines 2-cycle hit
L2 Unified	1MB; 4-way assoc., 64B lines 20-cycle hit
Memory Access Latency	151 cycles
Combined Branch Predictor	
Bimodal	8192 entries
Pag	8192 entries
Return Address Stack	64 entries
Branch Target Buffer	2048 entries; 4-way assoc.
Mispredict Latency	14 cycles

The seven benchmark-input pairs in Table 1.1 are simulated in *sim-outorder*. *sim-outorder* is modified to print simulation result for every sampling unit throughout the benchmark execution. Because cycle-accurate simulation is done throughout, there is no warm-up error in the experiment. Two sampling unit sizes are used: a sampling unit size of 1 million instructions for covering the range of 1 million to 100 million instructions, and a unit size of 100 million instructions for covering the range of 100 million to 5 billion instructions.

Figures 3.1 and 3.2 show the intracluster correlation coefficient for CPI for the two base sampling unit sizes. It is clear that different benchmarks show different intracluster correlation. However, except where two data points of *bzip2-source* in Figure 3.2 are very close to zero, all the intracluster correlation coefficients are positive. Based on Equation 3.5, the positive correlation means that using larger sampling units is not effective at improving accuracy.

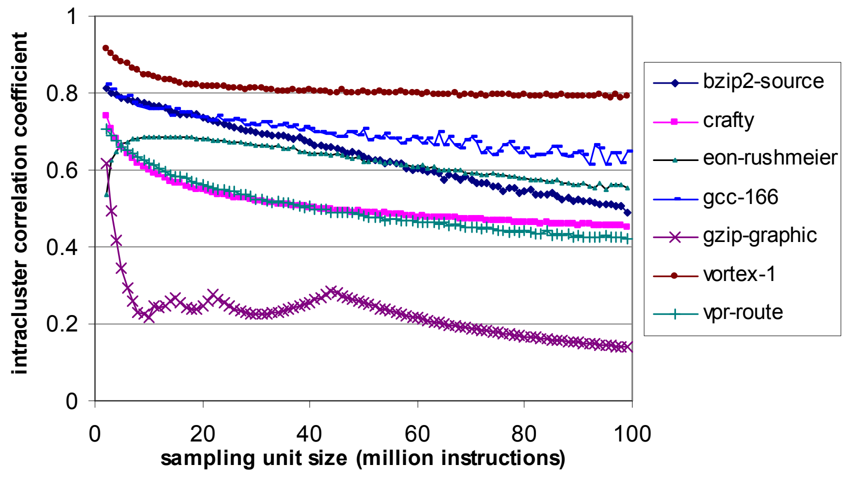

Figure 3.1: Intracluster correlation coefficient for CPI with baseline sampling unit size of 1 million instructions.

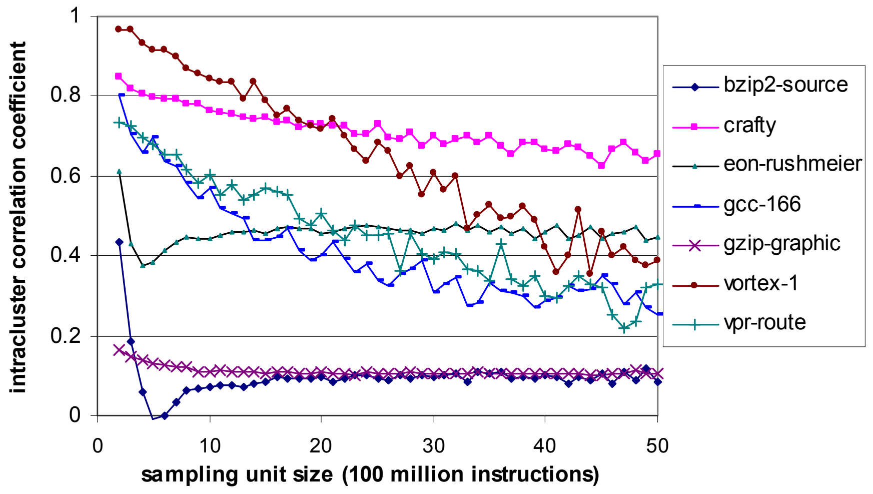

Figure 3.2: Intracluster correlation coefficient for CPI with baseline sampling unit size of 100 million instructions.

For verification, the standard deviation for different sampling unit sizes is calculated. The standard deviation, normalized to that of the original sampling unit size, is shown in Figures 3.3 and 3.4. The lowest smooth curve (with legend “org unit”) in the figure demonstrates the decrease of the standard deviation as the user takes more sampling units instead of using large sampling unit sizes. According to Equation 3.3, the standard deviation decreases with $1/\sqrt{M}$, and the curve is the same for all benchmarks. All the other curves show that the standard deviation generally drops as the sampling unit sizes are increased. Except for the two data points in *bzip2-source* in Figure 3.4, the curves do not drop as quickly as the “org unit” curve, which leads to the same conclusion as Figures 3.1 and 3.2 that larger sampling unit size is not effective at improving the accuracy of CPI. Consider *crafty* in Figure 3.4. Take a chunk of 100 million instructions as an example sampling unit because simulating a chunk of several hundred million of instructions is a popular practice. Suppose that the 95% confidence interval is *e* when simulating *crafty*. If the chunk size is increased to 1 billion instructions, then error can only be reduced to 0.89*e*, a marginal gain. On the other hand, the chunk size is kept at 100 million instructions, but 10 times more chunks are sampled, then the error limit is reduced to 0.32*e*, even though the total number of measured instructions remains the same.

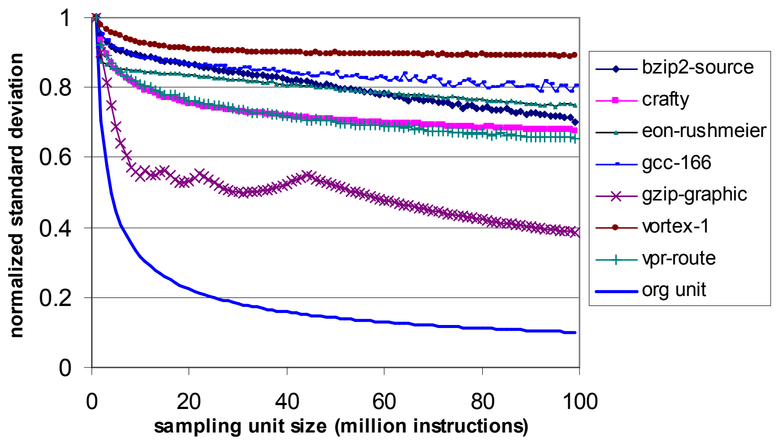

Figure 3.3: Normalized standard deviation for CPI with baseline sampling unit size of 1 million instructions.

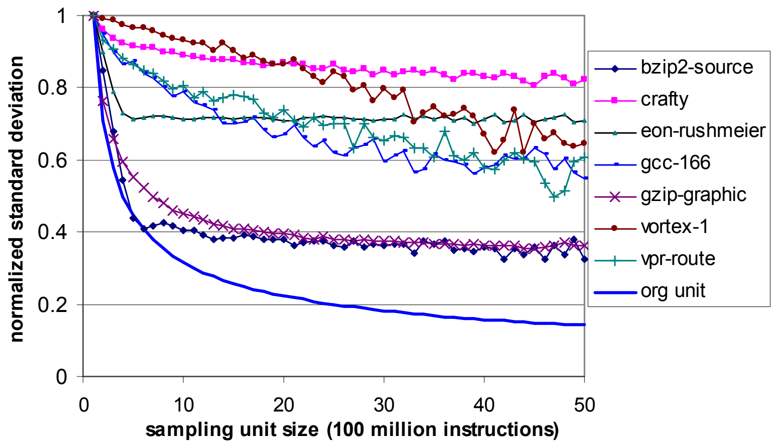

Figure 3.4: Normalized standard deviation for CPI with baseline sampling unit size of 100 million instructions.

Although sampling for CPI is the focus of this study, sampling is also useful for other metrics. Similar experiments are performed for the level one data cache misses per instruction and the branch misprediction per instruction with the two base sampling sizes. In most cases, a small sampling unit size is still more accurate than a large sampling unit size. Because these results are similar to those of CPI, they are not shown here. Only in two cases does a large sampling unit size results in smaller error. They are shown in Figure 3.5 (level one data cache misses per instruction for base sampling unit of 100 million instructions) and Figure 3.6 (branch misprediction per instruction for base sampling unit of 100 million instructions). In both figures, *bzip2-source* shows negative intracluster correlation coefficients. As verification, Figures 3.7 and 3.8 show the normalized standard deviation for the cache misses and branch misprediction. As the sampling unit size increases the standard deviation for *bzip2-source* drops quickly and soon goes below the “org unit” curve. For this particular benchmark, simulating a large chunk of 2 billion instructions gives better accuracy than using twenty chunks of 100 million instructions when measuring L1 data cache misses or branch misprediction.

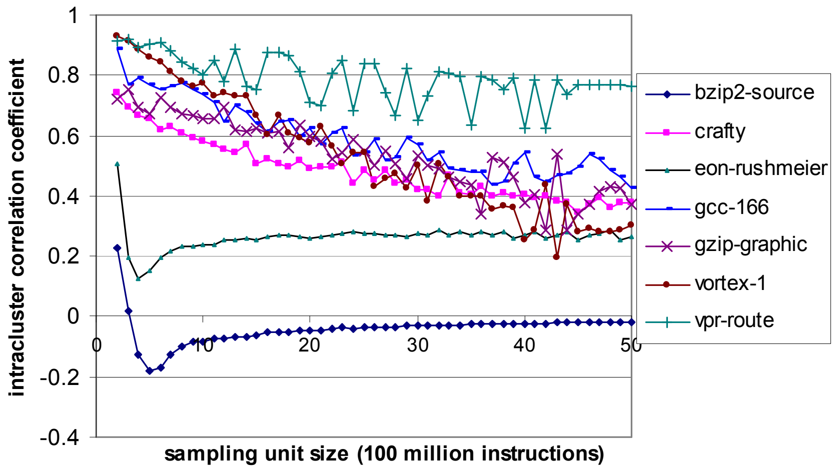

Figure 3.5: Intracluster correlation coefficient for L1 data cache misses with base sampling unit size of 100 million instructions.

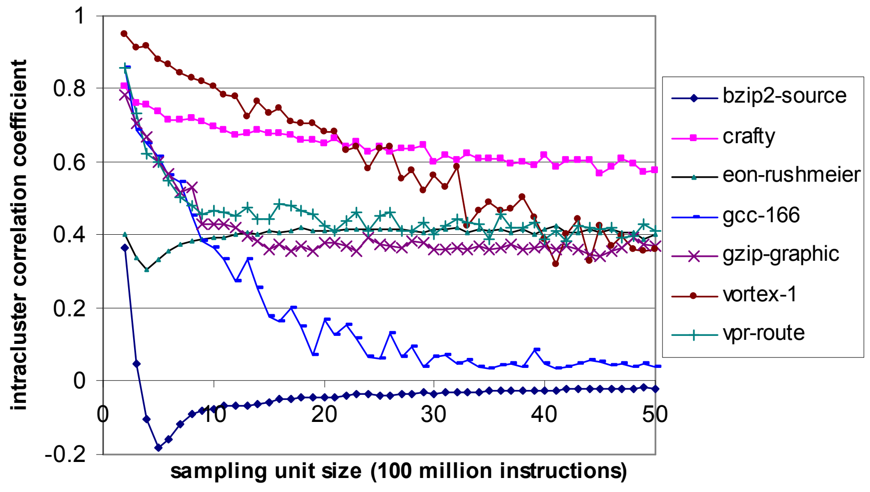

Figure 3.6: Intracluster correlation coefficient for branch misprediction with base sampling unit size of 100 million instructions.

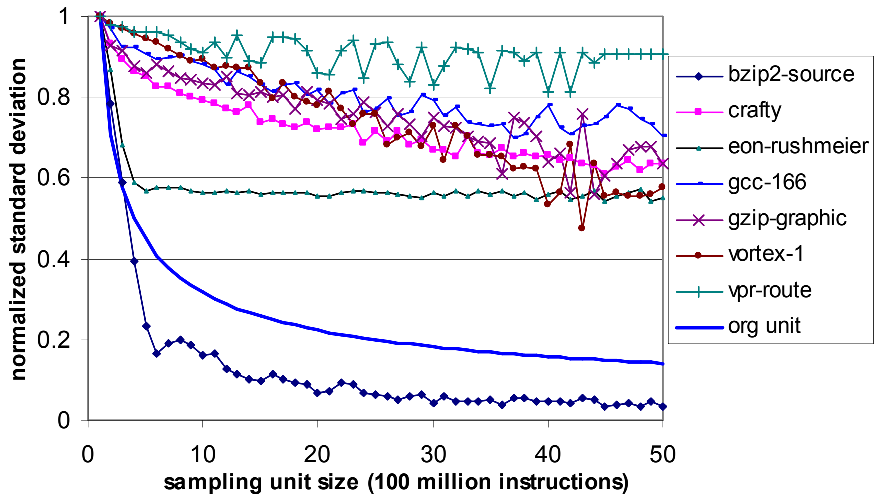

Figure 3.7: Normalized standard deviation for L1 data cache misses with base sampling unit size of 100 million instructions.

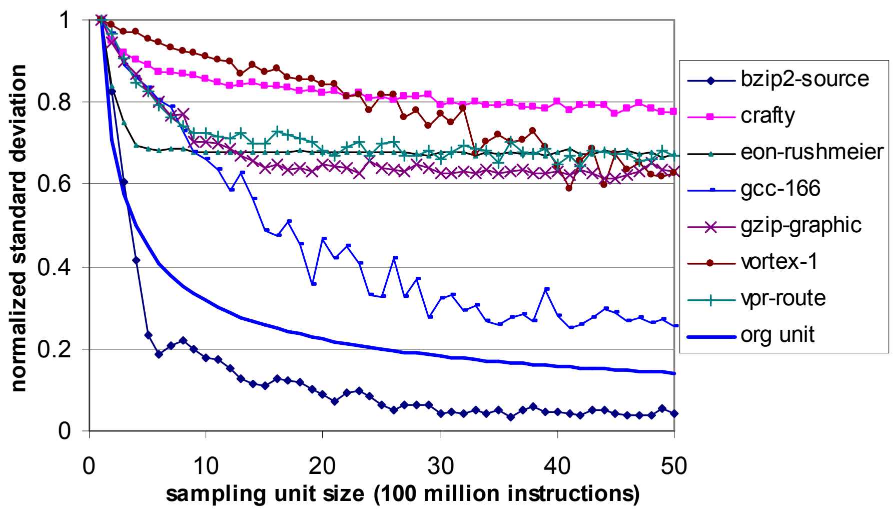

Figure 3.8: Normalized standard deviation for branch misprediction with base sampling unit size of 100 million instructions.

3.4 DISCUSSION

The results in the last section show that for most benchmarks and most metrics, using larger and larger sampling units is not a good way to improve the accuracy of the sampling result. A more effective way would be to keep the sampling unit size small but take more sampling units. There are, however, some benchmarks that favor large sampling units for some metrics. These experiments were done for only one processor configuration, so a natural question would be "is the positive intracluster correlation generally expected or is it a coincidence in the experiment?" Therefore, it is more important to discern the underlying reason than to merely present the observation. Finding out why benchmark *bzip2-source* produces peculiar results is also desirable.

Positive correlation inside a cluster means that the sampling units within one cluster exhibit similar metrics. One possible cause may be the phase behavior of some benchmarks. Within a phase, the program exhibits relatively constant behavior, so the metrics of the sampling units are very similar. However, a closer look shows that the positive intracluster correlation is more common than the phase behavior. For example, Figures 3.9 and 3.10 show the graph of CPI of every 100 million instructions for two benchmarks. Distinct phases can be observed in *vortex-1* whereas *crafty* looks like white noise to human eyes. But as shown in Figure 3.2, both benchmarks show highly positive intracluster correlation so there is probably a more general program property causing the positive correlation.

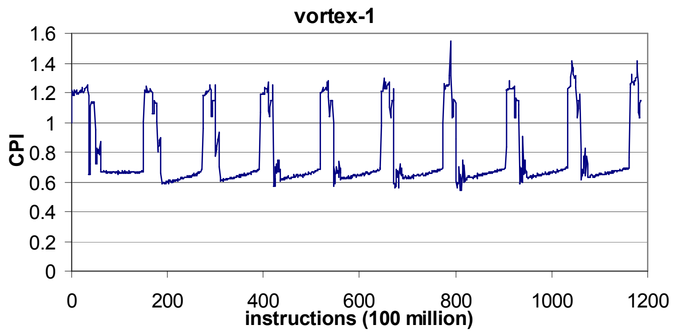

Figure 3.9: CPI for every 100 million instruction unit for *vortex-1*.

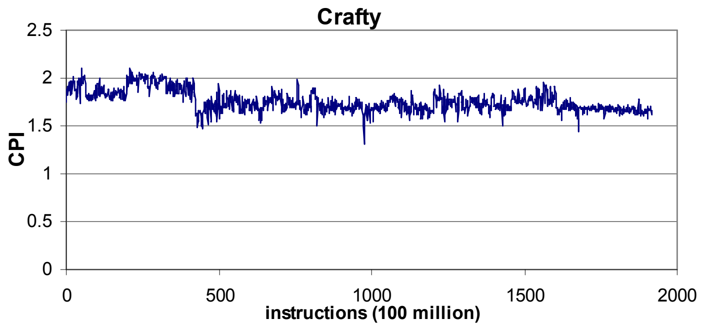

Figure 3.10: CPI for every 100 million instruction unit for *crafty*.

The positive intracluster correlation means that the sampling units in one cluster show similar metrics. However, the metrics are microarchitecture-dependent. To find out the underlying reason, the similarity between the sampling units themselves needs to be measured. One microarchitecture-independent metric to measure the similarity between instruction traces is the *Basic Block Vector* (BBV) proposed by Sherwood, *et al* [68]. A Basic Block Vector is a one-dimensional array with an element for each static basic block in the program. Each array element is the count of how many times a given basic block has been entered during a sampling unit or a cluster. The dissimilarity between the instruction traces in the two sampling units is reflected in the distance between the BBVs of two sampling units reflects. If the distance is small, then in the two sampling units the same set of basic blocks are executed with similar execution frequency. Thus the instruction traces in the two sampling units are similar. On the other hand, if the distance is large, then different set of basic blocks are executed or the same basic blocks are executed with very different frequency, so the two sampling units are dissimilar.

Let $\vec{v}_i$ denote the BBV for the *i*th sampling unit. The following metric is proposed to measure the average dissimilarity of two sampling units that are *h* sampling units apart.

$$d(h) = \frac{1}{N-h}\sum_{i=1}^{N-h} | \vec{v}_i - \vec{v}_{i+h} | \tag{3.7}$$

Obviously *d*(0) is always 0. To simplify calculation, random linear projection is used to reduce the dimensionality [14]. BBV distance is only comparable within a single benchmark. It makes no sense to compare the distance between benchmarks, so the distances are scaled in order to show them clearly in Figures 3.11 to 3.15[3]. One noticeable characteristic of the benchmarks is that the distance for most benchmarks shows a trend of going up, which means that the closer the two sampling units are in time, the more similar they are and vice versa. This is a type of general temporal locality

[3] The trend of the curves is important whereas the absolute value of y-axis is not meaningful.

of code. If a part of the program is executed in a sampling unit, then the same part will probably be executed in the near future (i.e. in the close neighbor sampling unit) in a similar fashion (i.e. with similar execution frequency of the basic blocks). In addition, Lau, *et al*. [39] has shown that if two sampling units are very similar in terms of BBV, then they will usually exhibit similar metrics such as CPI, cache miss rates, and branch misprediction rate. As a result, positive correlation among the nearby sampling units within a cluster is usually observed.

Larger sampling units are created by combining small sampling units that are close to each other into one sampling unit. These consecutive small sampling units are usually very similar and exhibit similar metrics because of the general code locality, so adding neighboring sampling units to the sample does not capture more information. The sampling units that are far apart are generally dissimilar. Therefore, using small sampling unit size and letting the units distributed throughout the whole instruction stream will give more information and the best accuracy.

However, there are notable exceptions to the general code locality. *Bzip2-source* with a sampling unit size of 100 million instructions shows an oscillating BBV distance graph, which means that the sampling units close to each other may be as different as the units that are far apart, so the only negative intracluster correlation occurs with *bzip2-source* in Figures 3.5 and 3.6. *Gzip-graphic* with sampling unit of 1 million instructions also shows a BBV distance graph that lacks general code locality but its intracluster correlation is still positive for various metrics. After all, the metrics such as CPI are microarchitecture-dependent. Even though the instruction streams in two sampling units are different, they may still exhibit somewhat similar CPI because the overall effect of the code on the microarchitecture may be similar.

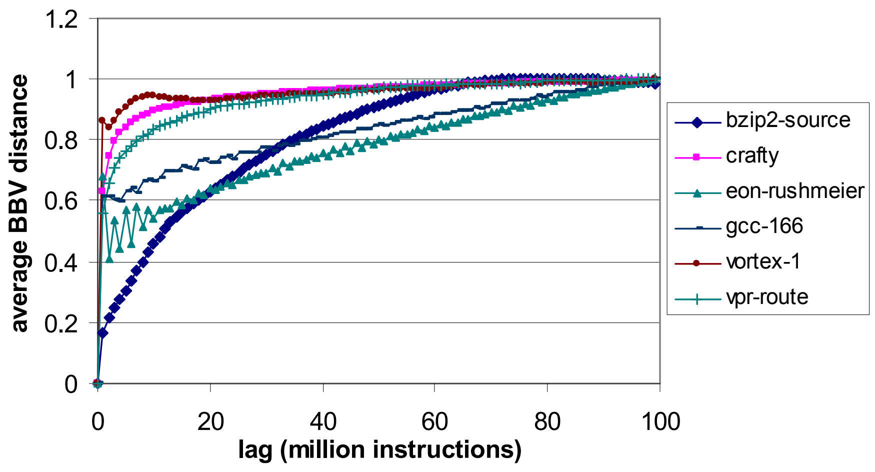

Figure 3.11: Average distance of BBV for sampling unit size of 1 million instructions for 6 benchmarks.

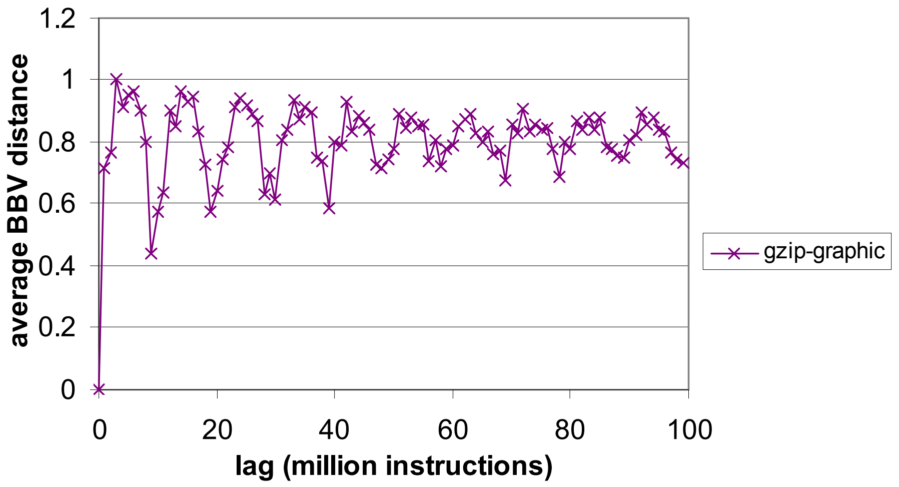

Figure 3.12: Average distance of BBV for sampling unit size of 1 million instructions for *gzip-graphic*.

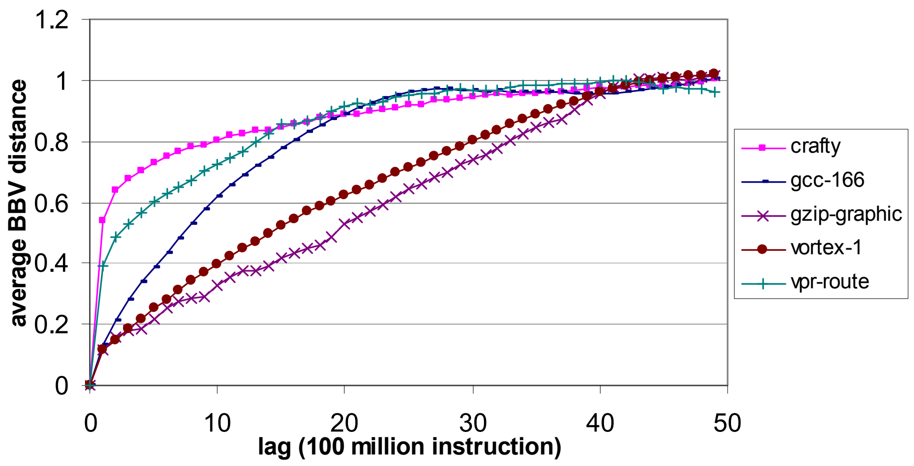

Figure 3.13: Average distance of BBV for sampling unit size of 100 million instructions for 5 benchmarks.

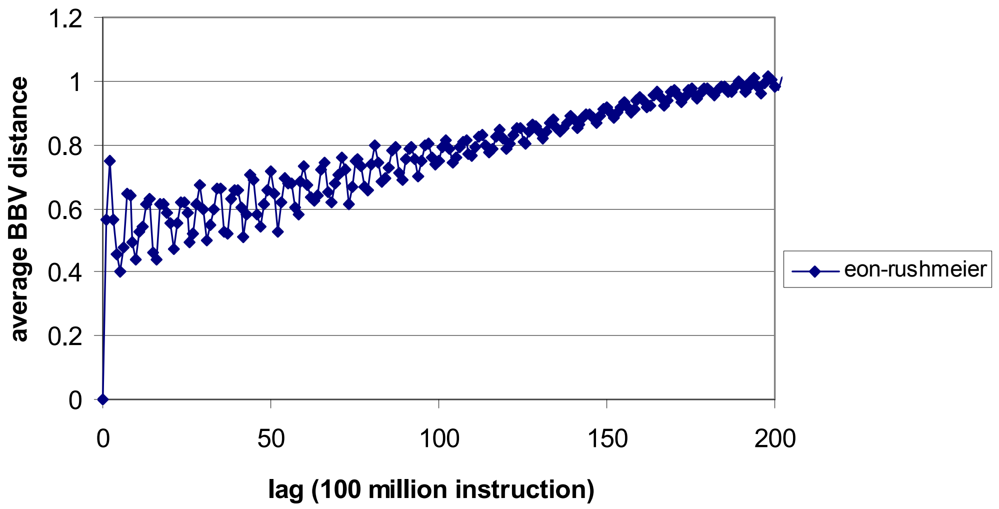

Figure 3.14: Average distance of BBV for sampling unit size of 100 million instructions for *eon-rushmeier*.

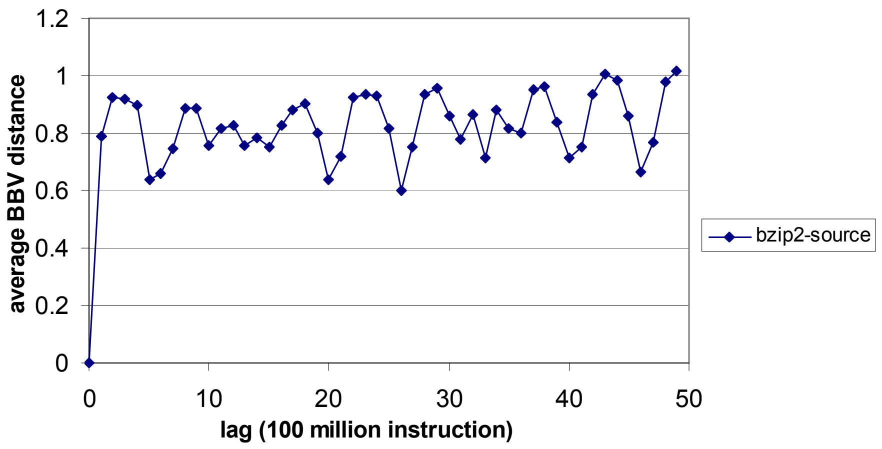

Figure 3.15: Average distance of BBV for sampling unit size of 100 million instructions for *bzip2-source.*

3.5 SUMMARY

Sampling is an effective technique to reduce simulation time while retaining good accuracy. Previously researchers have proposed different sampling techniques with vastly different sampling unit sizes. In practice, people are also using the one chunk sampling and simulating a larger and larger chunk, so it is unclear what a good sampling unit size should be.

In this chapter, the sampling unit size problem is studied by calculating the intracluster correlation coefficient. It is observed that most benchmarks exhibit positive intracluster correlation for various metrics in a wide range of sampling unit sizes. The positive intracluster correlation makes it less effective to use a large sampling unit size to improve the accuracy of sampling result.

The inherent characteristic of benchmarks that causes positive intracluster correlation and favors small sampling units is also investigated. Using the microarchitecture-independent BBV distance, it is shown that most benchmarks exhibit a type of generalized temporal locality. The instruction stream of a sampling unit is more similar to that of another unit that is nearby than to a unit that is far apart. Therefore, sampling units that are close by each other usually show similar metrics and thus are positively correlated. Grouping small sampling units into a large one is less accurate than distributing the small sampling units throughout the whole instruction stream.

Although uncommon, there are benchmarks that do not show the general code locality. These benchmarks are candidates for large sampling units. This lack of general code locality will not manifest as negative intracluster correlation in all cases, but some of the benchmarks do favor large sampling units for some metrics. Overall, however, using small sampling units is a safe bet unless there is clear evidence to show otherwise.

The conclusion that small sampling units are more effective at reducing sampling error is consistent with recent development in research on sampled microprocessor simulation. For example, in the precursor method to SimPoint, one large sampling unit of 300 million instructions is used. In SimPoint, sampling unit size is reduced to 100 million instructions. In the latest Variance and Early SimPoint, sampling unit size is further reduced to 1 million and 10 million instructions. The dramatic reduction of sampling unit size and the increase of sample size has significantly improved the accuracy of the SimPoint method. Although these sampling techniques are more sophisticated than simple random sampling and the authors did not discuss in detail the reason for the specific sampling unit sizes, the underlying reason for the diminishing sampling units appears to be the same as that elaborated in this chapter.

The conclusion in this chapter also argues against the popular one-chunk sampling, especially against the trend in which the chunk is becoming larger as better computers are available to run the simulation. Larger chunks will improve accuracy somewhat but are unlikely to be effective at it. A better way is to use smaller sampling units and distribute them throughout the whole benchmark execution.

Chapter 4. Sampling Techniques for Fast Simulation of OLTP Workloads

4.1 Introduction

Most of the research on sampling techniques for reducing simulation time has been focusing on SPEC CPU [75] benchmark suite, which consists of CPU-intensive programs with a single thread. Commercial workloads such as database systems are very important in the business world. They are known to exhibit characteristics drastically different from SPEC CPU programs [51][44][45], but the simulation methodology for commercial workloads has not been studied as thoroughly due to the complexity in setting up and tuning the workload. Online transaction processing (OLTP) is an important type of database workloads. It is the major task of traditional database systems. OLTP is critical for day-to-day business operations such as purchasing, inventory, and banking. The majority of transactions are short, fast updates and queries of a few records. ACID (Atomicity, Consistency, Isolation, and Durability) properties are fully enforced. In addition, OLTP applications are used interactively so the database must respond quickly.

OLTP workloads are very different from SPEC CPU benchmarks. SPEC CPU benchmarks are single-threaded CPU-intensive programs with little operating system or I/O activity. OLTP workloads, on the other hand, are multi-threaded and involve significant operating system and I/O operations. SPEC CPU programs perform a defined task. For example, the *gcc* benchmark analyzes the input source code and generates optimized Motorola 88100 assembly code. The performance metric is based on the time to complete the predefined task. In contrast, OLTP workloads do not have a predefined task. In a real business environment, OLTP workloads are supposed to run 24x7. The execution time of OLTP benchmarks is usually artificially limited so that the user can measure the performance in a reasonable time. Unlike SPEC CPU benchmarks, OLTP performance is measured using the throughput of transactions instead of the execution time of the program. After an OLTP workload is started, its performance changes in the beginning. Because an OLTP workload is considered non-terminating, the user cares only about its long-term performance. Therefore, the initial ramp-up period is deliberately ignored and only the throughput in steady state is measured. In comparison, the execution time of the entire benchmark in SPEC CPU is included in the performance metric.

The problem of long simulation time is even more serious for OLTP workloads. Consider TPC-C [77], the most widely used OLTP benchmark. The benchmark is required to run for at least 2 hours. In comparison, the execution of a SPEC CPU2000 benchmark on a modern computer usually takes minutes. Despite the seriousness of the problem and the uniqueness of OLTP workloads, most simulation time reduction techniques are proposed for and validated against the SPEC CPU benchmark suite.

Sampling is the most widely used simulation time reduction technique. Most of the sampling techniques fall into two categories. In the first type of technique, the sampling units are picked randomly or with equal gap (i.e. systematic sampling). Because of the autocorrelation between the sampling units in the instruction, small sampling unit size is preferred for efficiency (see Chapter 3). A large number of sampling units are commonly measured to achieve good accuracy, usually expressed in terms of a confidence interval. SMARTS is a recently proposed techniques that uses systematic sampling [81]. It has chosen a sampling unit size of 1,000 instructions for SPEC CPU2000 benchmarks. These sampling units are distributed evenly throughout the entire instruction stream. The simulator switches between detailed mode (only for selected sampling units) and functional warm-up mode (for all remainder of the instruction stream). In the detailed mode, the complete microarchitecture is simulated to get the CPI. In the functional warm-up mode only the caches and the branch predictor are simulated to speedup the simulation while keeping the warm-up error at minimum. Usually tens of thousands of sampling units are simulated in the cycle-accurate mode. It has been shown that on average, CPI for a whole benchmark can be estimated to within an error of 3% with 99.7% confidence by measuring fewer than 50 million instructions.

The other sampling technique, which I call representative sampling, differs philosophically. Instead of randomly sampling, a few relatively large chunks of instructions are carefully, yet automatically, selected to represent the whole instruction stream. It utilizes the well-observed phase behavior of program execution. A phase can be defined as a portion of dynamic execution of a program for which most of the performance metrics such as CPI, show very little variance. Because the performance metrics remain stable in a phase, simulating only one chunk of instructions from each phase can greatly reduce simulation time with little loss of simulation information. The SimPoint technique family is the best-known representative sampling techniques [61][69]. In SimPoint, the entire instructions streams are divided into chunks of 100 million instructions [69] (In newly proposed versions, it has been reduced to 1 or 10 million instructions [61]). The Basic Block Vector (BBV) is recorded for each chunk. Then cluster analysis is performed to group the chunks into clusters. The chunks in the same cluster have similar BBVs and thus similar CPIs. The chunk with BBV closest to the center of cluster is selected from each cluster to represent the entire instruction stream. These chunks are called simulation points.

Both types of technique have been successfully applied to SPEC CPU2000. There is no single sampling technique that is the best for all situations. Each has its own advantages and weaknesses. The user should select the sampling technique based on the simulation infrastructure, the characteristics of the workload, and the tradeoff he or she is willing to make. SMARTS can give a confidence interval to quantify the error of the result. For simulating OLTP workload, only one checkpoint at the beginning of the steady state is needed. Therefore, the storage cost is minimal. However, it requires that the simulator be able to switch modes during the simulation. Not every simulator has this capability, and the user may not want to significantly modify a simulator after it has been developed and validated. Furthermore, the efficiency of SMARTS depends on the characteristics of the workload and the relative speed of the two modes. If the variation in the program execution is very small and speed difference between functional warm-up and detailed simulation is not large enough, then the functional warm-up will become a significant bottleneck even though only a tiny fraction of the instruction stream needs to be simulated in detailed mode.

Although a confidence interval is unavailable in SimPoint, it works very well in most cases. SimPoint is also faster than SMARTS for SPEC CPU2000 benchmarks especially when checkpoints are used. However, it does sometimes result in large errors for some benchmarks on some microarchitecture configurations. For SPEC CPU2000, simulator switching modes can also be used for SimPoint. The simulator enters detailed simulation just before each simulation point, whose beginning point can be specified by the number of instructions from the beginning of the benchmark execution. For OLTP benchmarks, the changes in microarchitecture affect the execution path of the program [3]. Therefore, the simulation points can no longer be marked in this way. A checkpoint has to be used for each simulation point. In the simulator used in this study, even with compressed incremental checkpointing, each checkpoint may require up to 300MB space. As a result, the user has to trade storage cost for faster simulation.

Both of the techniques have only been evaluated for SPEC CPU-like benchmarks. Their applicability to OLTP workloads has not been investigated despite the uniqueness of OLTP workloads. In this chapter, both sampling techniques are evaluated for an OLTP benchmark. More importantly, by taking advantage of the characteristics of OLTP workloads, a dynamic stopping rule for simple random sampling is designed. The dynamic stopping rule is very easy to use and more efficient than SMARTS.

The chapter is structured as follows. The experimental setup is described in Section 4.2. The applicability of SimPoint is studied in Section 4.3. In Section 4.4, the dynamic stopping rule for simple random sampling is proposed. Section 4.5 summarizes the chapter.

4.2 Experimental Setup

The DBT-2 benchmark developed by Open Source Development Labs is used [57] for experiments in this chapter. DBT-2 is a derivative of the TPC-C benchmark. It emulates the database system of a whole parts supplier company operating out of a number warehouses. Each warehouse supplies 10 sales districts, each of which has 3,000 customers. Each warehouse tries to maintain stock for the 100,000 items in the company's catalog. 10% of the orders sent to a warehouse must be supplied by other warehouses. There are a total of 5 types of transactions: New-Order, Payment, Order-Status, Delivery, and Stock-Level.

DBT-2 was developed for the Linux operating system. For this research it was ported to Solaris 9. The database system used is PostgreSQL. PostgreSQL creates one process for each client connection so it cannot efficiently support a large number of simultaneous connections. Therefore, DBT-2 is run in a 3-tier mode in the experiment. The terminal drivers do not directly connect to the database server. Instead, they connect to the client in the mid-tier, which acts as a connection collector. Five warehouses are created. The total database data size is about 1GB including the index and other meta-data. The client generates 5 threads and maintains 5 connections to the server. The Remote Terminal Emulator (RTE) emulates 50 terminals, which simultaneously generate transaction requests to the client. The think time and keying time are set to zero so that the terminals send requests as quickly as possible to keep the server fully utilized. The RTE and the client are running on the same simulated machine as the database server. The setup is verified on a real SPARC machine. The total CPU utilizatioin of the RTEs and the client is only about 1%. Therefore, the effect of running the RTEs and the client on the same machine with the server can be ignored.

Simics [52], a commercial full system simulator, is used to simulate a SunFire server running Solaris 9. The clock frequency of the processor is set to 2GHz. Two configurations with different cache latencies are modeled as shown in Table 4.1. For OLTP workloads, the user only cares about the steady state performance. Therefore, the system has been warmed up with about 3,000 transactions before starting measurement.

Table 4.1: System configurations

		Configuration 1	**Configuration 2**
Processor		Clock frequency 2GHz Fetches, executes, retires, and commits 4 instructions per cycle	
L1 instruction cache	Organization	32KB 8-way set associative, 64 bytes/line	
	Latency	1 cycle	2 cycle
L1 data cache	Organization	32KB 8-way set associative, 64 bytes/line	
	Latency	1 cycle	2 cycles
L2 cache	Organization	2MB 8-way set associative, 64 bytes/line	
	Latency	6 cycles	9 cycles
Memory		2 GB, 200 cycles latency	2 GB, 250 cycles latency

4.3 APPLYING SIMPOINT TO OLTP BENCHMARK

The SimPoint methodology is popularly used to reduce the simulation time for SPEC CPU, but its applicability to OLTP workloads needs further investigation. Patil, *et al.* applied SimPoint methodology to large commercial applications running on Intel Itanium machines [59]. However, most of their programs were run in single-threaded mode. They evaluated one multi-threaded benchmark suite, SPEC COMP2001, and concluded that it was hard to apply SimPoint method in their experiments. They ran instrumented programs on real machines to collect the BBV profile. But the execution of multi-threaded program are non-repeatable on real machines so it was extremely difficult to repeat the run to obtain the trace of simulation points after the cluster analysis of the BBV profile. In this chapter, a full system simulator is used, which gives deterministic and repeatable results; so this study does not have the same problem.

The efficacy of SimPoint depends on the existence of homogeneous phases in the benchmark and the ability of the cluster analysis to find them. Figure 4.1 shows the CPI for every 1 million instructions from the benchmark running on configuration 1. To make the graph legible, a total of only 5 billion instructions are shown. However, the complete experiment result, which is much longer, is not very different. Unlike many SPEC CPU programs, there are no obvious large-scale phases. This is consistent with the common understanding of the benchmarks. Most programs in SPEC CPU accomplish one task and do it in several steps, resulting in large-scale phases, each of which usually corresponds to one such step. For example, *gzip* alternates between compressing and decompressing the input file, producing distinct compression and decompression phases. Readers are referred to [69] for more examples of large-scale phases in SPEC CPU programs. In contrast, The TPC-C benchmark consists of only 5 different types of transactions. Most of the transactions in OLTP are relatively short, and there are a number of transactions going on simultaneously, which are driven by a random number generator. At any moment, these simultaneous transactions may be at different stage of execution. The interleaving of the transactions depends on the operating system scheduling. Therefore, it is less likely to see many long, contiguous phases.

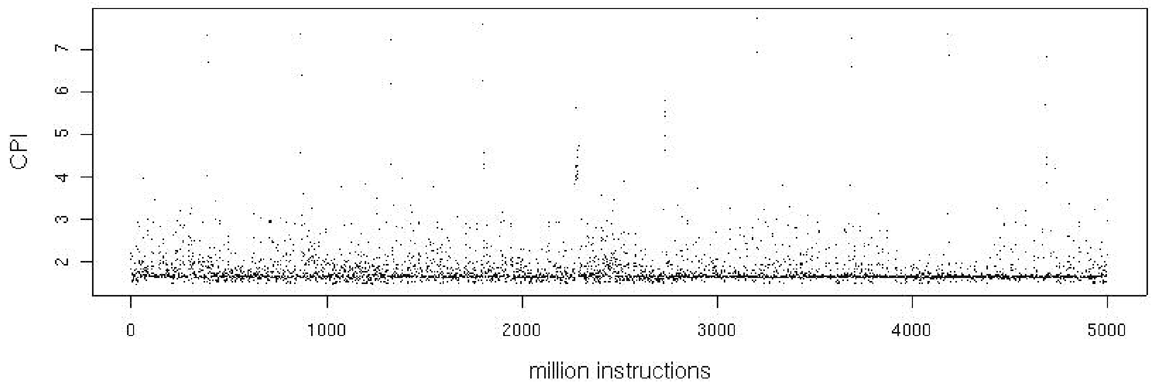

Figure 4.1: CPI for every million instructions on configuration 1.

The lack of large-scale phases does not preclude the applicability of SimPoint because a phase may consist of many small non-contiguous parts and eludes visual inspection. To evaluate the effectiveness of SimPoint, it is compared with simple random sampling. As an example, suppose 7 chunks are simulated in SimPoint. To gauge the efficacy of SimPoint, 7 chunks are randomly picked, and their mean CPI and the error with respect to the full detailed simulation are calculated. The random sampling is repeated 1,000 times to get the average error. Then this average error of random sampling is compared to the error in SimPoint. Three different chunk sizes are studied: 1 million instructions, 10 million instructions (both used in Variance SimPoint), and 100 million instructions (used in the original SimPoint). To mitigate the cost of storing checkpoint files, number of clusters between 2 and 20 are considered. Figures 4.2 to 4.7 compare the error in SimPoint with the average error for random sampling. All results assume perfect warm-up.

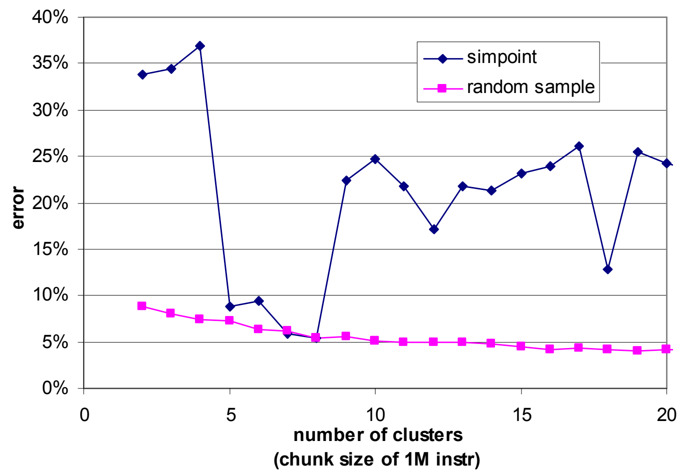

Figure 4.2: Error for SimPoint and simple random sampling on configuration 1 with chunk size of 1 million instructions.

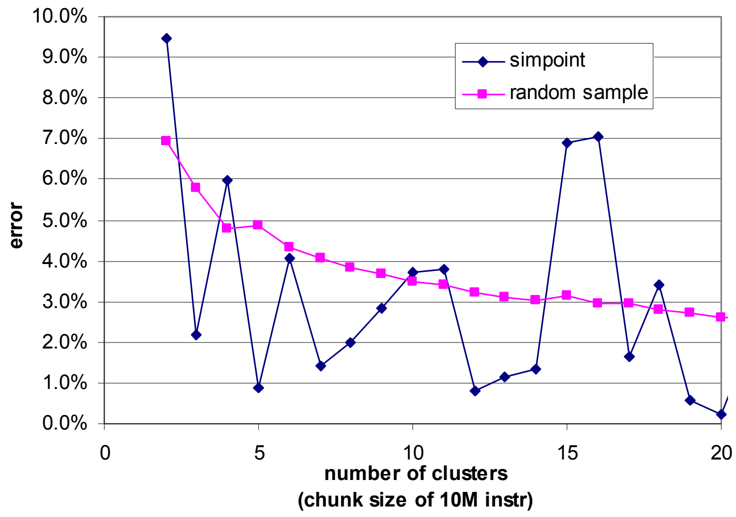

Figure 4.3: Error for SimPoint and simple random sampling on configuration 1 with chunk size of 10 million instructions.

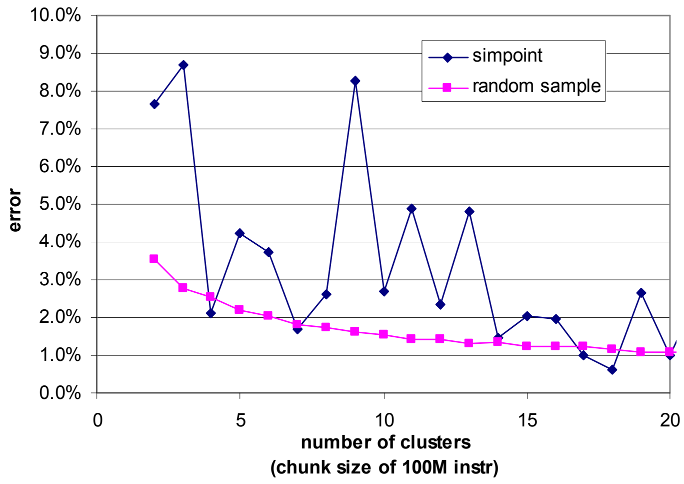

Figure 4.4: Error for SimPoint and simple random sampling on configuration 1 with chunk size of 100 million instructions.

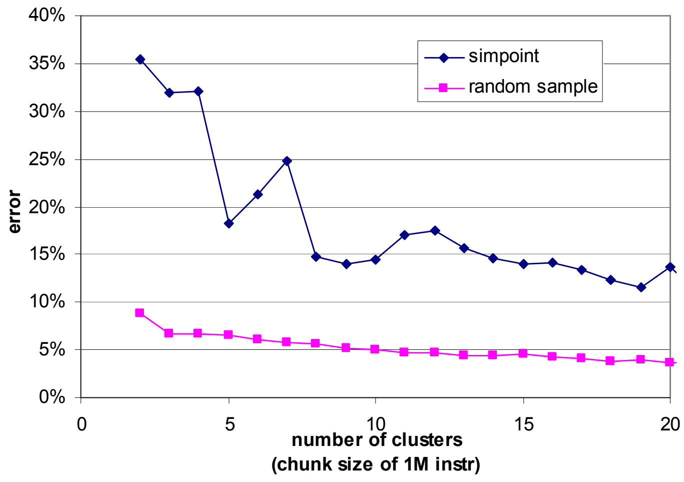

Figure 4.5: Error for SimPoint and simple random sampling on configuration 2 with chunk size of 1 million instructions.

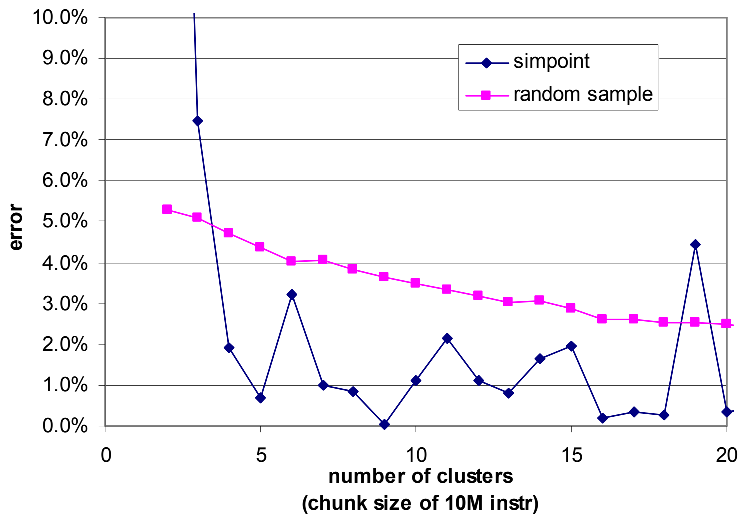

Figure 4.6: Error for SimPoint and simple random sampling on configuration 2 with chunk size of 10 million instructions.

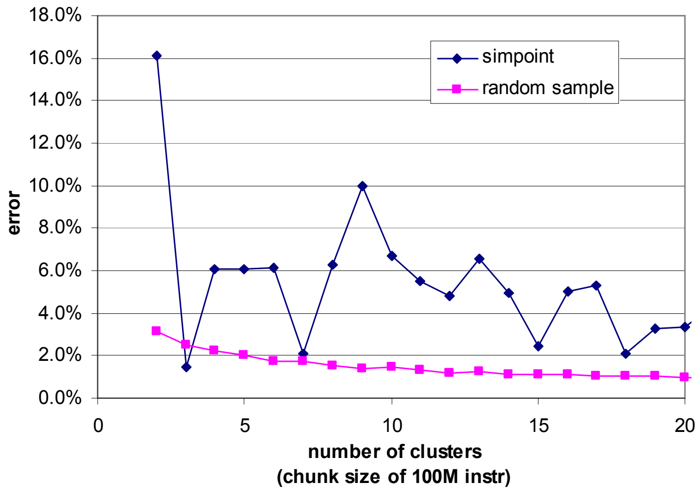

Figure 4.7: Error for SimPoint and simple random sampling on configuration 2 with chunk size of 100 million instructions.

An important observation is that the efficacy of SimPoint is dependent on the chunk size. For chunk size of 1 million instructions and 100 million instructions, on average SimPoint is not as accurate as simple random sampling. But for chunk size of 10 million instructions, SimPoint gives much better accuracy on both configurations. Therefore, SimPoint is applicable to the simulation of OLTP workloads, but to take advantage of SimPoint the user has to carefully choose the chunk size. With a good chunk size it can give much more accurate result in comparison to simple random sampling.

4.4 Applying Simple Random Sampling to OLTP Workloads

4.4.1 Selecting Sampling Unit Size

Simple random sampling has been successfully used to reduce the simulation time of SPEC CPU2000 programs [81][12]. It is the simplest form of sampling, and therefore has the fewest limitations and the widest applicability. In addition, simple random sampling has the solid foundation of statistical sampling theory, which can give the user a confidence interval to quantify the sampling error without simulating the entire benchmark in the detailed mode. To use simple random sampling, one has to decide on the sampling unit size first. Chapter 3 has shown that for SPEC CPU benchmarks, small sampling units usually give better accuracy than large sampling units when the same number of instructions are simulated in the detailed mode. The same method, which relies on the sign of the intracluster coefficient, is adopted here to evaluate different sampling unit sizes for the OLTP benchmark.

As in SMARTS, the sampling unit size of 1,000 instructions is used as the baseline. The intracluster correlation coefficients are calculated from the experiment data. The result is shown in Figure 4.8. It is clear that all the correlation is positive. Therefore, using sampling unit size of 1,000 instructions is better than any sampling unit size between 1,000 instructions and 100,000 instructions. Evaluation of larger sampling unit size up to 100 million instructions gives the same conclusion. Smaller sampling unit sizes are not studied because with too small unit size, the overhead of detailed warm up can increase quickly. In addition, the sampling unit is measured in the number of instructions, but on a superscalar machine, multiple instructions can be committed in one clock cycle. Therefore, at the beginning and the end of a sampling unit, partial cycles have to be counted. For example, suppose that one sampling unit starts from instruction #1000, but instructions #998 to #1001 are committed in one cycle. Then the fraction of cycle for instructions #1000 and #1001 has to be counted if very small sampling unit sizes are used. With sampling unit size of 1,000 instructions such fraction cycles can be ignored.

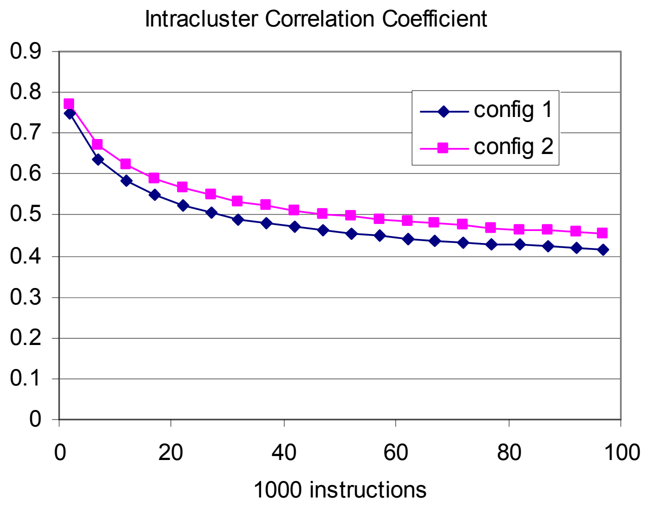

Figure 4.8: Intracluster correlation coefficient with the baseline sampling unit size of 1000 instructions.

4.4.2 Dynamic Stopping Rule for Simulating OLTP Workloads

In random sampling techniques for SPEC CPU such as SMARTS, before the simulation the user specifies target accuracy expressed as the relative error limit at a certain confidence level. Then the user gives an initial sample size based on previous experience or guess, and runs the simulation with this sample size. After the simulation, a confidence interval is calculated and compared to the target accuracy. If the confidence interval does not meet the target accuracy, then a new sample size is calculated based on the result from the first simulation. A second sampled simulation is run with the new sample size. The result of the second simulation is expected to meet the user's target accuracy. Therefore, two simulations are usually needed to measure the CPI for one benchmark.

It is desirable to eliminate the second simulation. This can be achieved by taking advantage of the characteristics of OLTP workloads. As has been discussed before, a major difference between SPEC CPU programs and OLTP benchmarks is that a SPEC CPU program usually accomplishes one task and does it in multiple steps whereas an OLTP benchmark processes multiple sequences of transaction requests from different terminals simultaneously. The sequences of transaction requests are randomly generated. Therefore, in the steady state the workload reaches a statistical state of equilibrium. Take any two large enough chunks of instruction streams and inspect the execution characteristics (e.g., CPI) within the chunk. The CPI graphs will look random and different in the two chunks, but their statistical properties are the same. They will exhibit the same mean, the same variance, and the same autocorrelation function. Because of this special characteristic of OLTP workloads, the second simulation in SMARTS can be eliminated. A dynamic stopping rule is proposed, in which the user monitors the current confidence interval during the simulation and stops the simulation once the target accuracy has been met. The following is the procedure.

1. Before the simulation, the user specifies the target accuracy.
2. The user chooses a sampling rate (i.e. sampling 1 unit out of *n* units).

3. Start the simulation. As in SMARTS, the simulator only simulates the units that are sampled in detailed mode and measures its CPI (detailed simulation). Two thousand instructions before the sampling unit are also simulated in detail, but the CPI is not measured (detailed warm-up). Only the caches and the branch predictor are simulated for all the instructions between the sampling units (functional warm-up).
4. Once the simulation has passed the minimum simulation length of L_{min} instructions, start calculating the confidence interval after simulating each sampling units. If the confidence interval meets the target accuracy, stop. Otherwise, continue simulation.

The user should do some initial experiments to choose the sampling rate. Too low a sampling rate will result in unnecessarily long simulation. Although the number of instructions in detailed simulation remains largely unchanged, the total number of instructions in functional warm-up will increase. The minimum simulation length should be set so that the mean CPI of the consecutive L_{min} instructions chosen from anywhere in the steady state execution should remain relatively constant.

This dynamic stopping rule has several advantages. It requires only one checkpoint. Thus the storage cost is minimum. More importantly, it needs only one-pass simulation, which increases the usability and reduces the simulation time. In SMARTS if the user's initial choice of sample size is a little below optimal for a SPEC CPU program, then the confidence interval will be slightly larger. Only a few more sampling units are needed, but the user has to simulate the benchmark again to meet the target accuracy. With the dynamic stopping rule for OLTP benchmarks, the user just continues stimulating until the target accuracy is met, which is a much shorter additional simulation than doing the whole benchmark again.

Experiments are conducted to evaluate the dynamic stopping rule for OLTP workloads. The sampling unit size of 1,000 instructions and the sampling rate of 1/1,000 are used. Figure 4.9 shows the relative error limit at 99% confidence level as the simulation progresses on the two configurations. At the beginning, the error limit is not stable, moving up and down rampantly. This is because the sample size in the beginning is too small to give an accurate estimation of the confidence interval. In addition, DBT-2 benchmark itself exhibits large variability at small granularity: Even if all the instructions in a short interval were simulated in detail, the CPI could still be significantly different from the long-run mean CPI. Therefore, the minimum simulation length should be at least 500 million instructions. And a larger number should be used for better result.

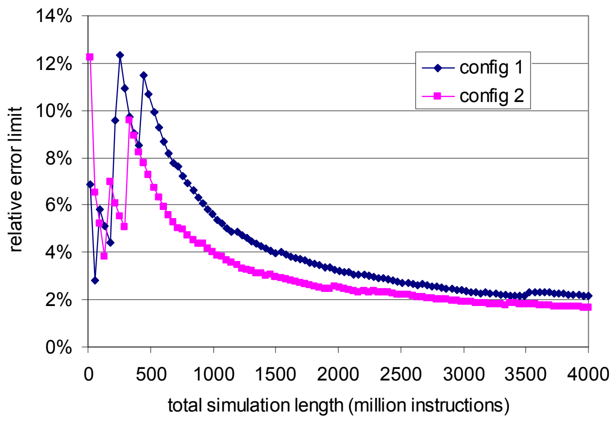

Figure 4.9: Relative error limit at 99% confidence level as the simulation proceeds.

Suppose the user want the error to be within 3% at 99% confidence level. Targt accuracy of 3% is chosen as the example because TPC-C allows up to 2% variation in reported throughput (Clause 5.5.1). Therefore, target accuracy below 2% would be unnecessary. Table 4.2 shows the result of applying dynamic stopping rule on the two configurations. A total of 2.28 and 1.50 billion instructions are simulated on the two configurations, but the number of instructions simulated in detailed mode is three orders of magnitude smaller. Also, the real error lies within the confidence interval.

Table 4.2: Result of applying dynamic stopping rule.

	Configuration 1	Configuration 2
Sample size	2282	1502
Total simulation length (million instructions)	2282	1502
Instructions simulated in detail (million)	6.85	4.51
Real error	2.73%	1.62%

4.5 Summary

OLTP workloads are important in the business world and they have very distinct characteristics. However, most of the simulation time reduction techniques are designed for SPEC CPU type of programs. The applicability of these techniques to OLTP workloads needs validation. In addition, new simulation time reduction techniques can be created for OLTP workloads by taking advantage of their characteristics. In this study, two sampling techniques (SimPoint and SMARTS) are applied to the simulation of OLTP workloads. It is found that OLTP workloads do not exhibit long consecutive phases because it executes a randomly generated mixture of simultaneous database transactions. Nevertheless, SimPoint is applicable to OLTP workloads. But its efficacy depends on the chunk size. If a good chunk size is chosen, it is much more accurate than simple random sampling with the same chunk size and the same simulation length. Therefore, when using SimPoint, the user should carefully choose the chunk size.

Simple random sampling such as SMARTS is also good at reducing simulation time for OLTP workloads. By utilizing the stationary characteristics of OLTP workloads, a dynamic stopping rule is proposed for random sampling. The simulator calculates the current confidence interval while it is doing the simulation. The simulator stops once the confidence interval meets the user's target accuracy. This method obviates the second simulation in SMARTS for SPEC CPU programs. It improves the usability and reduces the simulation time.

The contribution of this chapter is primarily in the observations and the methodology. But the exact numbers should be taken very cautiously because the current setup is too small compared to an authentic TPC-C setup that has been audited and approved by TPC. For example, the optimal sampling unit size in the user's setup may be very different from ours (10 million instructions). Nevertheless, the observation of the dependence of SimPoint's efficacy on the chunk size as is still valid. Therefore, the user should search for good chunk size when using SimPoint.

Chapter 5. Efficiently Evaluating Performance Improvement In Sampled Processor Simulation

5.1 Introduction

There has been extensive study on sampled processor simulation, but all previous research focuses on the accuracy of CPI or IPC [76][24][81][12][31][61][68][69][71][30][42]. However, the goal of a simulation is usually to evaluate the benefit of some microarchitectural enhancement, in which case, the absolute value of CPI may not be overly important. Instead, an accurate estimate of the relative performance improvement is more desirable. The term "speedup" is used to evaluate the benefit of microarchitecture enhancement. Speedup, denoted R, is defined as the ratio of the execution times before and after the microarchitectural enhancement when the same benchmark is run. Assuming that the clock frequency remains the same, it is also the ratio of the CPI's before and after the enhancement.

The user not only wants to estimate the value of the speedup but also wants to determine the error in the estimation. The error can be quantified with a limit on relative error[4], which can be converted from the confidence interval. No previous research has given a method to calculate the confidence interval for speedup. The SMARTS simulation technique proposed by Wunderlich, *et al.* can provide a confidence interval for CPI [81]. A straightforward method is to run SMARTS on the baseline and improved configurations to get two confidence intervals for CPI and then calculate the confidence interval for speedup using interval arithmetic. This method is hereafter called the *interval arithmetic method*. Suppose that the 95% confidence intervals before and after the

[4] Relative error for speedup is $|R_{sample}-R_{real}|/R_{real}$. When speedup is large the user may be willing to tolerate larger absolute error in speedup, so the relative error is used instead of the absolute error in this chapter.

microarchitectural enhancement are 0.820±0.020 and 0.617±0.015, which corresponds to a ±2.5% error. One may compute the confidence interval for speedup to be (0.800/0.632, 0.840/0.602), a relative error of ±5%. This example shows that constraining the relative error in speedup to be within *e* would require the error in CPI to be within *e*/2. Because at a given confidence level the confidence interval is inversely proportional to the square root of the sample size, reducing the relative error limit in half would require simulating 4 times more instructions in detail.

However, as will be demonstrated later, this estimation of error limit is too pessimistic and there is a better way to quantify the error in speedup. In the next section the ratio estimator in the sampling theory is introduced and a new sampling method is proposed to calculate the speedup and its error limit. The merit of the method is experimentally verified in Section 5.3. Section 5.4 summarizes this chapter.

5.2 Evaluating Performance Improvement with Ratio Estimator

The ratio estimator in the sampling theory calculates the ratio of two population means from a sample [10]. For each sampling unit, there are two characteristics, y_i and x_i (i=1, 2, ..., N). A random sample of size n is taken and y_i and x_i of each sampled unit (i=1, 2, ..., n) are measured. The goal is to estimate R, the ratio of the population mean of y to the population mean of x ($R = \bar{Y} / \bar{X} = \sum_{i=1}^{N} y_i \Big/ \sum_{i=1}^{N} x_i$). Based on the sampling theory, R is estimated as

$$\hat{R} = \frac{\bar{y}}{\bar{x}} = \sum_{i=1}^{n} y_i \Big/ \sum_{i=1}^{n} x_i \tag{5.1}$$

Its variance is estimated as

$$v(\hat{R}) = \frac{(1-f)}{n\bar{x}^2}(s_y^2 + \hat{R}^2 s_x^2 - 2\hat{R} s_{yx}), \tag{5.2}$$

where

$$s_{yx} = \frac{\sum_{i=1}^{n}(y_i - \bar{y})(x_i - \bar{x})}{n-1} \tag{5.3}$$

and S_y and S_x are the sample standard deviation of *y* and *x*. If the sample is large enough so that the normal approximation applies, the confidence interval for *R* at the confidence level of 1-α can be obtained as

$$(\hat{R} - z_{1-\alpha/2}\sqrt{v(\hat{R})}, \hat{R} + z_{1-\alpha/2}\sqrt{v(\hat{R})}). \tag{5.4}$$

where $z_{1-\alpha/2}$ is the (1-α/2) quantile of a unit normal distribution. Similarly, given a relative error limit of e at a confidence level of 1-α, the required sample size is

$$n = \frac{z_{1-\alpha/2}}{e\bar{y}}\sqrt{(1-f)(s_y^2 + \hat{R}^2 s_x^2 - 2\hat{R}s_{yx})} \tag{5.5}$$

If for each sampling unit, x_i and y_i are the CPI of the unit before and after the microarchitectural enhancement, then *R* is the speedup. The estimation of speedup (Equation 5.1) is quite intuitive and is the same as in the interval arithmetic method, but the calculation of the confidence interval is completely different. Based on the above theory, the following general procedure is proposed to calculate the speedup and to quantify its error [49].

1. Before the measurement, the user sets a target accuracy expressed as a relative error limit *e* at a certain confidence level 1-α.
2. Divide the full instruction stream into *N* chunks of *m* continuous instructions. Take a systematic sample or random sample of size *n*.
3. Measure the CPI of each sampled unit before the microarchitectural enhancement. Record all the CPIs (x_i).
4. Measure the CPI of the *same* sampled units after the enhancement. Record all the CPIs (y_i).
5. Calculate the speedup and its relative error limit or confidence interval with Equations 5.1 through 5.4. If the error limit meets the target accuracy, then stop. Otherwise, calculate the new sample size from Equation 5.5 and repeat step 3 and 4.

The key point in the procedure is to make sure that the same sampled units are measured in the two simulation steps (steps 3 and 4 above). But the instruction stream may be different in each run of the same benchmark. For a user mode simulator like SimpleScalar [6], this is caused by operating system calls (e.g *gettimeofday*) returning different result in each run. For example, in two runs of *gcc-166*, the difference in the number of dynamic instructions was 332,372. Although this difference only accounted for 0.00071% of the total instructions executed, it would cause different units to be sampled in the two runs if a small sampling unit size is used (e.g., 100-10,000 instructions as in SMARTS [81]). To solve this problem, the user must make sure that the dynamic instruction stream in each run is exactly the same. In the experiment, the *eio* trace is captured with SimpleScalar *sim-eio* utility. Then all the benchmark programs are run with the *eio* trace to guarantee the same instruction sequence.

Simulating a superscalar microprocessor, which can commit more than one instruction in a cycle, may raise a subtle problem. For example, the user may select (committed) instructions #201 to #400 as one sampling unit. Suppose instructions #198 to #202 are committed in one cycle. Then two instructions at the beginning of the sampling unit (#201, #202) only constitute partial cycle. To accurately measure the CPI for this sampling unit, one has to accurately count these partial cycles. To avoid dealing with the partial cycles, sampling unit size above 10,000 instructions is used in the experiment so that the partial cycles can be ignored.

5.3 Experiments and Results

Experiments are conducted to study the application of ratio estimator in sampled processor simulation. The procedure in the previous section does not specify how to measure the CPI of each sampling unit. It can be done by simulating the complete benchmark and switching between cycle accurate mode and functional simulation mode

[81][31]. Or it can be done by checkpointing the state before each sampling unit and simulating each checkpoint directly. The first method is used in the experiment. Caches and branch predictors are continuously warmed up functionally to approximate perfect warm-up [81]. Four thousand instructions before every sampling unit are simulated with cycle accurate simulator to warm up other microarchitectural structures. An 8-way and a 16-way out-of-order superscalar processor are simulated to calculate the speedup. The microarchitecture configurations given in Table 5.1 are adapted from [81].

Table 5.1: Processor configurations.

Parameter	**8-way (baseline)**	**16-way**
Machine Width	8	16
RUU/LSQ size	128/64	256/128
Memory System	32KB 2-way L1 I & D, 2 ports, Unified 1M 4-way L2	64KB 2-way L1 I & D, 4 ports, Unified 2M 8-way L2
ITLB / DTLB	4-way 128 entries 4-way 256 entries 200 cycle miss penalty	4-way 128 entries 4-way 256 entries 200 cycle miss penalty
L1/L2/Memory Latency	1/12/100 cycles	1/16/100 cycles
Functional Units	4 I-ALU 2 I-MUL/DIV 2 FP-ALU 1 FP-MUL/DIV	16 I-ALU 8 I-MUL/DIV 8 FP-ALU 4 FP-MUL/DIV
Branch Predictor	Combined 2K tables 7 cycle misprediction penalty 1 prediction/cycle	Combined 8K tables 10 cycle misprediction penalty 2 predictions/cycle

Eight benchmarks from SPEC CPU 2000 are simulated in a modified SimpleScalar 3.0 *sim-outorder*simulator. Three sampling unit sizes are used: 10,000 instructions, 1 million instructions, and 10 million instructions. Wunderlich, *et al.* [81] reports that the optimal sampling unit size is in the range of 100 to 10,000 instructions in their experiment setup. The size of 10,000 instructions from their study is chosen here. Unless the user handles partial cycles accurately as discussed in the previous section, using smaller sampling units may increase measurement error. Also as discussed later, warm-up error becomes pronounced with small sampling units for some benchmarks. Sampling unit sizes of 1 million and 10 million instructions are used in the latest Variance and Early SimPoint method [61]. The initial sample size of 3,000 is used for sampling unit size of 10,000 instructions, and initial sample size of 1,000 for sampling unit size of 1 million and 10 million instructions.

As an example the relative error limit of 2% at 95% confidence level is set as the target accuracy. After the initial sampling the sample size required to achieve the target accuracy for speedup is calculated based on Equation 5.5. For comparison, the sample size required to achieve the same target accuracy for CPI is also calculated (see [81] for the equations). The result is drawn in Figures 5.1 to 5.3. It can be seen that for most benchmarks the sample size for measuring speedup is only a small fraction of that for measuring CPI. To more accurately quantify the phenomenon, the ratio of sample size for speedup to the sample size for CPI is calculated for the 16-way issue processor. The geometric mean of the ratio for the benchmarks is 0.127 (for Figure 5.1), 0.107 (for Figure 5.2) and 0.115 (for Figure 5.3). Therefore, it is more cost effective to directly measure the speedup than measure CPI. In other words, to achieve a relative error limit of 2% on the speedup, users do not need to estimate CPI to the same relative error limit. Instead, they can measure, on average, only 1/9 of the instructions that are required for estimating CPI to the same relative error limit. Comparing with the interval arithmetic method, the saving is even more. The interval arithmetic method yields the same value of speedup but is far too pessimistic in quantifying its error and requires very unnecessarily large sample size. As discussed in Section 5.1, the interval arithmetic method would require that the relative error limit for the CPI be reduced to half of 2%, which will in turn quadruple the sample size. In the experiment configuration, using ratio estimator technique the geometric mean of the reduction in sample size will be about 36X compared to the interval arithmetic method. Therefore, using the ratio estimator equations can significantly reduce the sample size. In addition, using different target accuracy will not change savings of the proposed method because varying the target accuracy will equally affect the ratio estimator method and the arithmetic interval method.

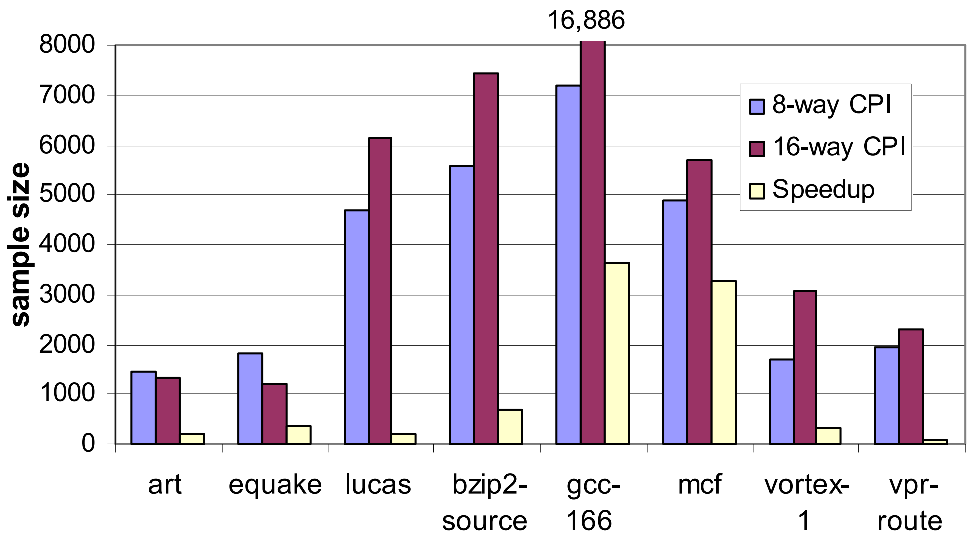

Figure 5.1: Sample sizes required to achieve relative error limit of 2% at the confidence level of 95% for sampling unit size of 10,000 instructions.

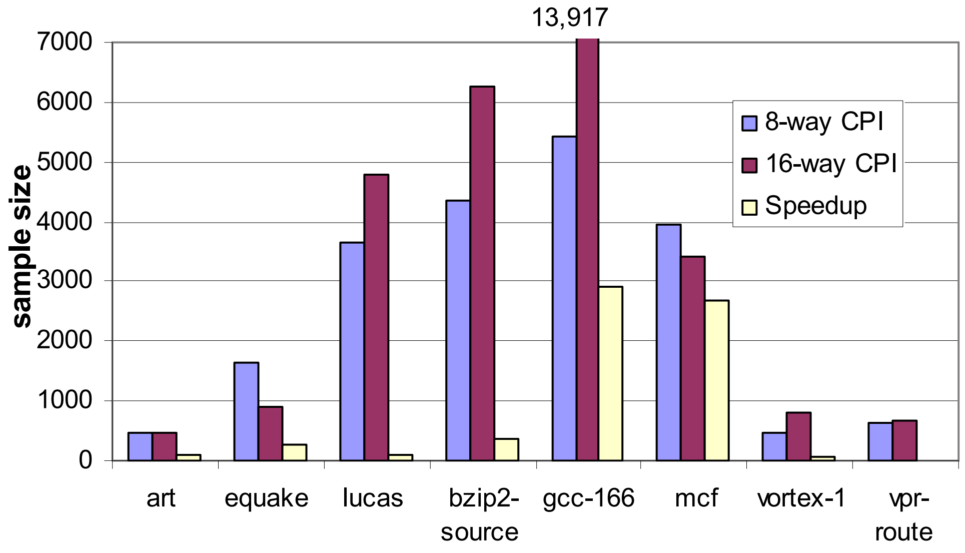

Figure 5.2: Sample sizes required to achieve relative error limit of 2% at the confidence level of 95% for sampling unit size of 1 million instructions.

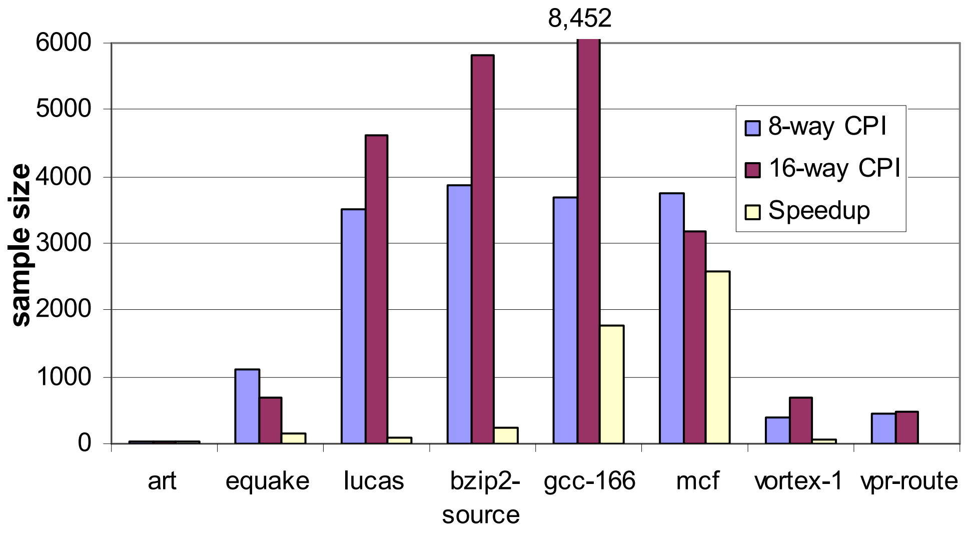

Figure 5.3: Sample sizes required to achieve relative error limit of 2% at the confidence level of 95% for sampling unit size of 10 million instructions.

There is no reason to doubt the ratio estimator theory, but it is still desirable to experimentally verify that the computed sample size does meet the target accuracy. The cycle accurate simulator *sim-outorder* is modified to dump the CPI for every sampling unit in the benchmark execution so that the population and the real speedup value R_{real} can be obtained. Monte Carlo method is employed to validate the target accuracy. Suppose the computed sample size is *n*. A random sample of size of *n* is taken from the population and the speedup from this sample is computed according to Equation 5.1. Then another random sample of size *n* is taken from the population and the speedup is computed again. The process is repeated many times (10,000 times in the experiment). If 95% of these speedup values lie within ±2% of the real value R_{real}, then the computed sample size *n* meets the target accuracy. If, on the other hand, a much lower percentage of speedup values are within the relative error limit, then the computed sample size is too small and a larger sample size is required.

Verification is not done for the sampling unit size of 10,000 instructions. Firstly, the data set of population is large, thus difficult to process. Secondly, with sampling unit size of 10,000 instructions the warm-up error is not negligible for some benchmarks. For example, according to Equation 5.4, with 99% confidence he (relative) sampling error in speedup for *vortex-1* should be within 0.88% but the actual relative error is 1.72%. Thus the majority of the error should be from warm-up. Therefore, verification is not done for this sampling unit size. The verification results for sampling unit sizes of 1 million and 10 million instructions are shown in Table 5.2. Columns 2 and 4 show the sample size calculated from Equation 5.5, which is the same as in Figures 5.2 and 5.3. Columns 3 and 5 give the percentage of the speedup values from the Monte Carlo experiment that lie within the relative error limit of ±2%. Ideally, this percentage should exactly be 95%. There is inevitably some error in Monte Carlo experiment, and the ratio estimator itself has slight bias (see [10] for detail), so the percentage numbers in the table are not exactly 95%. However, none of the number is much lower than 95%. Therefore, it can be concluded from the verification result that the sample size computed from the ratio estimator theory does satisfy the target accuracy requirement.

Table 5.2: Verification of the sample size computed from ratio estimator theory.

Benchmark	**Sampling unit size of 1 million instructions**		**Sampling unit size of 10 million instructions**	
	Sample size	**Percentage within error limit**	**Sample size**	**Percentage within error limit**
art	101	95.0%	24	95.4%
equake	254	94.7%	147	95.3%
lucas	87	94.7%	84	94.8%
bzip2-source	359	95.0%	254	95.2%
gcc-166	2902	96.0%	1769	98.8%
mcf	2694	95.7%	2587	99.0%
vortex-1	76	95.2%	60	95.2%
vpr-route	12	94.8%	8	95.3%

The above result may be surprising at the first glance. How could the speedup be more accurate than the CPIs from which it is computed? This is because the two configurations being evaluated are usually not fundamentally different. The CPI values may vary widely during the benchmark execution, but if the CPI is high (or low) for one part of the code on the first configuration, then the CPI for this part of code is probably also high (or low) on the second configuration. Thus the CPIs are usually correlated and the ratio of the two CPIs (i.e. the speedup) does not change as much as the CPIs themselves. At a given accuracy, the sample size is largely determined by the variation normalized to the mean (i.e. the coefficient of variation). The small variation in the speedup leads to the small sample size. Figure 5.4 shows a graph of CPIs and the speedup for a small portion in the execution of *vortex-1*. All the metrics (CPIs and speedup) are normalized to their respective mean so that the normalized variation can be compared. It is obvious that the CPIs on the two configurations follow each other and the variation in the speedup is much smaller than in the CPIs[5]. The speedup has a smaller normalized variation and thus requires smaller sample size.

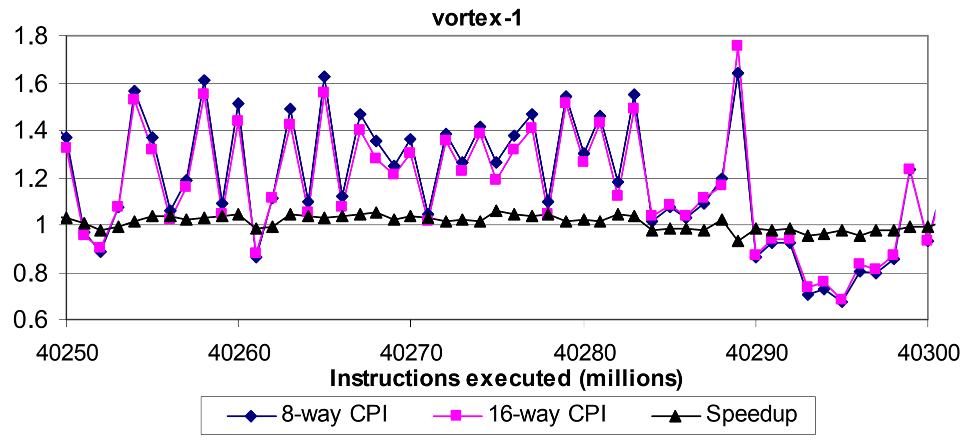

Figure 5.4: Normalized CPIs and speedup. Each data point is for one million instructions.

5.4 SUMMARY

Simulation of processors is mostly used for evaluating the benefit of some microarchitectural enhancement, in which the speedup is a more important metric than the CPI. In this chapter the ratio estimator method from sampling theory is applied to sampled processor simulation to quantify the error of speedup. To achieve a given error limit for speedup, it is not necessary to estimate CPI to the same accuracy. For the same relative error limit, measuring speedup requires fewer instructions to be simulated than measuring CPI. In the experiments, using the ratio estimator to estimate speedup results

[5] Please note that the absolute speedup is about 1.5. The impression that the CPIs for the two configurations are almost the same is just an artifact of normalization.

in a sample size 9X smaller than estimating CPI, and 36X smaller than estimating speedup using interval arithmetic.

This technique has great potential to reduce simulation time for computer architects. Especially when checkpoint files or trace files are used and each sampling unit is simulated directly with explicit warm up, the reduction in the sample size will directly translate into savings in storage space and simulation time. A 9X smaller sample size will result in 9X shorter simulation.

In real-world simulations, the microarchitectural changes are often smaller than those in the experiment in the previous section, so the CPIs may be even more correlated and the ratio estimator technique is probably highly effective. However, it is important in future work to explore the limitation of the technique and to provide guidelines to assess its effectiveness if drastically different configurations are being compared.

Chapter 6. SMA: A Self-Monitored Adaptive Cache Warm-Up Scheme for Microprocessor Simulation

6.1 Introduction

In a sampled simulation the original full instruction stream is divided into non-overlapping chunks of continuous instructions. Recently, Wunderlich, *et al.* applied sampling theory to microarchitecture simulation [81]. They showed that CPI could be estimated to within an error of 3% with 99.7% confidence by measuring fewer than 50 million instructions per benchmark. This accounts for only 0.029% of the average dynamic instructions executed for a benchmark program. It appears that sampling has effectively solved the problem of prohibitively long simulation times.

The aforementioned results are obtained under the assumption of ideal warm-up or perfect initial state before each sampling unit. As expected, the CPI of each sampling unit depends not only on the instructions executed in the unit, but also on the initial state of all microarchitectural structures at the beginning of this unit. Executing a limited number of instructions before a sampling unit to get the (approximately) correct initial state is known as *warming up* the microarchitecture. The number of instructions used for warm-up before a sampling unit is its *warm-up length*. For small structures like the ROB, the reservation station, and the register file, thousands of instructions are enough to put them into the correct state. However, some structures in the microprocessor like the branch target buffer and the caches can hold thousands to millions of bytes. It is difficult to ensure that they are in the correct initial state before every sampling unit in the simulation. If the initial state is not correct, the error can be large. For example, Haskins, *et al.* reported that ignoring warm-up in their experiment could result in an error as high as 15% in simulated CPI [29]. Thus adequate warm-up is critical to the accuracy of sampled simulation. Warm-up not only affects accuracy but also incurs overhead and increases simulation time. When simulating a processor with large caches, a large number of instructions may be needed for adequate warm-up, which prolongs the simulation. Therefore, the warm-up issue is very important in sampled simulation. A good warm-up scheme should achieve a desired level of accuracy while devoting as few instructions as possible for warm-up.

Warm-up is still an important issue in sampled microprocessor simulation and deserves further research. But there is an opinion that warm-up is largely solved and little reduction in simulation time can be accomplished with better warm-up techniques. For example, MRRL is claimed to have achieved 90% of the maximum possible simulation speed [29]. However, careful analysis of the experiment reveals the functional simulation as the bottleneck because every benchmark is simulated functionally from beginning to end. In simulation environment in this study, the relative speed of functional simulation (no microarchitecture simulation at all), functional warm-up (only cache and branch predictor simulation), and cycle-accurate simulation, is 1:1/2.8:1/16[6]. Fifty 1 million-instruction sampling units are used in the MRRL paper [29]. Suppose that a benchmark is 100 billion instructions long and on average each sampling unit needs 30 million instructions for warm-up[7]. Then the percentages of time spent in functional simulation, functional warm-up and cycle-accurate simulation are 95.17%, 4.06%, and 0.77%. It is obvious that the functional simulation is the bottleneck, so even getting rid of warm-up overhead altogether will provide little benefit. However, if users save the checkpoints or the traces for each sampling unit, they no longer needs to run the benchmark from beginning to end and is able to simulate for each sampling unit directly.

6 The relative speed numbers are highly dependent on the simulator and the configuration being simulated.

7 No warm-up length number is given in the MRRL paper [29]. This number is based on the experiment with MRRL in Section 3.4.2.

In this case, removing the warm-up overhead will give 6.25 times speedup in simulation! Therefore, a better warm-up technique is still highly desired.

It would be desirable for a warm-up scheme to be adaptive to cache sizes and benchmark variability characteristics. Intuitively, small caches do not need the same warm-up lengths as large caches. Similarly, programs with different variability/phase behavior would benefit from a scheme that adapts to the program. In this chapter, the warm-up process of the processor caches is studied and a self-monitored adaptive warm-up scheme for simulation is proposed. The simulator monitors the warm-up process of the caches and determines whether the caches are warmed up based on simple heuristics. Unlike previous research, this method is both adaptive to the characteristics of the benchmark and the cache configuration being simulated. The details of the proposed adaptive warm-up scheme are presented. The new method is compared with the best warm-up schemes from prior research. Overall, the proposed scheme achieves very good accuracy with lower warm-up overhead than previously proposed techniques.

This chapter is structured as follows. To overcome the weaknesses of previous warm-up methods, the new self-monitored adaptive cache warm-up scheme is proposed in Section 6.2. The proposed technique is evaluated experimentally in Section 6.3. This chapter is summarized in Section 6.4.

6.2 SMA: A SELF-MONITORED ADAPTIVE WARM-UP SCHEME

MRRL and BLRL are the two most recently proposed cache warm-up techniques (see Chapter 2). One major problem with MRRL and BLRL is that they do not take the cache configuration into account. Ideally, the cache warm-up process depends on both the workload and the cache organization. A small direct mapped cache is intuitively easier to warm up than a large highly associative one, but both MRRL and BLRL methods call for the same warm-up length given the same p-value. Therefore, using any fixed *p*-value in the techniques may result in under-warm-up or over-warm-up for different caches.

Careful examination of the previous techniques shows that they are not warm-up methods per se. The caches are warmed up through simulating instructions before each sampling units. All the methods just help the user to decide when the warm-up is enough, so why not monitor the warm-up process in the simulator to decide whether the warm-up is enough? This is exactly the rationale behind the proposed self-monitored adaptive (SMA) warm-up technique [47][48].

In SMA warm-up, as in the previous techniques, the simulator does functional warm-up before switching to detailed cycle-accurate simulation. During the functional warm-up, the caches are accessed but no pipeline stages are simulated. The warm-up process of the cache is monitored. The simulator switches to cycle-accurate simulation as soon as the cache is deemed "warmed up". Therefore, the warm-up length is not fixed but adaptive. Unlike previous approaches, this technique implicitly considers both the workload characteristics and the cache organization. Fewer instructions will be used for warming up a small direct-mapped cache than for a large highly associative one.

To monitor the cache warm-up process, all the cache blocks are initialized to the *cold-start* state before the functional warm-up. The address/tag in a cold-start block is unknown because it depends on the previous instructions, which were not simulated. When a cache access is initiated, the set index to the cache can be calculated. If the memory address is not found in this set and one or more cache blocks in this set is in the cold-start state, then the cache access is called a *cold-start access*. It is not known whether a cold-start access will result in a cache miss or a hit. When data is brought to a cold-start state cache block, the block changes to the *valid*state. Once a cache block leaves the cold-start state, it never goes back to this state again. Any state other than the cold-start state is called a *known* state.

Two aspects of the warm-up process are monitored. Firstly, the simulator keeps track of the percentage of cache blocks in the cold-start state. This number monotonically decreases during warm-up. If no cache block is in cold-start state, the cache is completely warmed up. So the outcome of every future reference is guaranteed to be known. Secondly, the simulator monitors the number of cold-start accesses during an interval. When the cache is large, or the working set of the program is small, it may take too long to completely warm up the cache. In this case, the cache is deemed warmed up when the number of cold-start accesses is below a user-defined threshold. Unlike a completely warmed up cache, there is no guarantee that all future references will access blocks in known state. However, the possibility of cold-start accesses is low. The detailed information on choice of parameters for the interval size and threshold is given in the next section. Monitoring the warm-up process is a very low overhead operation, it only increments or decrements a couple of counters at a cold-start cache access. There is no time overhead for accessing cache blocks in known state. The number of cold-start accesses usually decreases quickly.

Another problem with the previous methods is that users generally do not know how accurate the warm-up was after the simulation. They have to rely on previously published validated results. However, the user's configuration may not be the same as in the published paper. SMA can give users some indication of the accuracy of the warm-up after the simulation. After switching to cycle-accurate simulation, the simulator continues to count the number of cold-start accesses. In this way, after the simulation the user knows how much of all the cache misses are due to cold-start accesses. In the experiment a cold-start access is treated as a cache miss. So the number of cold-start accesses is usually the upper bound of the overestimation of cache misses. For example, if during cycle accurate simulation of 1 million instructions the user only sees 20 cold-start cache references, then he or she knows that the overestimation of cache misses is very unlikely to go above 20 and the CPI result should be fairly accurate.

6.3 EXPERIMENTS AND RESULTS

Ten benchmarks from SPEC INT2000, listed in Table 6.1, are used in the experiment. The programs, downloaded from the SimpleScalar web site [70], are compiled for the Alpha ISA. Table 6.2 shows the main processor configuration used in the experiment. This configuration is adapted from the SMARTS paper [81].

Table 6.1: Benchmarks, their data set and dynamic instruction count. The data set name is appended to the benchmark name.

Benchmark	**# of Instructions (million)**
gcc-166	46, 918
bzip2-source	108,878
crafty	191,883
eon-cook	80,614
gap	269,036
gzip-graphic	103,706
mcf	61,867
twolf	346,485
vortex-1	118,977
vpr-route	84,069

Table 6.2: Processor configuration.

Parameter	8-way (baseline)
Machine Width	8
RUU/LSQ size	128/64
Memory System	32KB 2-way L1 I & D, 2 ports, Unified 1M 4-way L2
ITLB / DTLB	4-way 128 entries 4-way 256 entries 200 cycle miss penalty
L1/L2/Memory Latency	1/12/100 cycles
Functional Units	4 I-ALU 2 I-MUL/DIV 2 FP-ALU 1 FP-MUL/DIV
Branch Predictor	Combined 2K tables 7 cycle misprediction penalty 1 prediction/cycle

6.3.1 Variability in Warm-Up Process

Much research has been done on devising and comparing warm-up techniques, but few of the projects shed light on the warm-up process itself. In this research, experiments are conducted to study how the cache warm-up process proceeds. Only one important issue in cache warm-up, the variability in the warm-up length, is presented here. The effectiveness of the new warm-up technique depends on the variability. If a constant warm-up length is good for all situations, then the PRIME method with fixed warm-up length will be the best. However, if the required warm-up length changes widely, then a good warm-up technique needs to adapt to all the factors that affect the warm-up process.

In the experiment, each benchmark execution is divided into *segments* of 100 million instructions. To study the cache warm-up process of each segment the simulator sets all cache blocks to the cold-start state at the beginning of each segment. The warm-up process in each segment is tracked. For the L1 data cache, the warm-up length for each segment needed to put every cache block into the known state is recorded. Table 6.3 lists the mean and the standard deviation of the warm-up length per sampling unit. The L2 cache may not completely warm up at the end of 100 million instruction segments, so the warm-up length needed to warm up 50% of the cache blocks is recorded instead. The statistics for the L2 cache warm-up length is shown in Table 6.4. These warm-up lengths are not the warm-up lengths required for sampled simulation. Nevertheless, they reflect the large variability in the warm-up process. The results clear show that the warm-up length is different for different benchmarks. It is also widely different within one benchmark. In many cases the standard deviation of the warm-up length of different segments is as large as the mean. Therefore, devoting a fixed number of instructions to warm-up as in PRIME method is not a good idea.

Comparing Table 6.3 and Table 6.4, it is also observed that to warm-up a certain percentage of the cache blocks, the large L2 cache needs much longer warm-up than the small L1 cache. Figure 6.1 contrasts the different warm-up requirement of L1 and L2 caches from another angle. It shows the number of cold-start cache accesses per 100,000 instructions as the caches are being warmed up for all the benchmarks. Except for at the beginning part of *eon-rushmeier*, the cold-start cache accesses for the L2 cache decreases much slower than for the L1 cache for all benchmarks. It is clear from these graphs that a processor with only the L1 cache requires much shorter warm-up than a processor with both L1 and L2 caches. Therefore, it is important for a good warm-up method to also take the cache configuration into consideration. MRRL and BLRL both adapt to the different segment in a benchmark but they do not adapt to the cache configuration.

Table 6.3: Warm-up length for warming up all cache blocks in L1 data cache (in 100,000 instructions).

Benchmark	Mean	Standard Deviation	Max	Min
bzip2-source	17.80	16.82	184	1
gcc-166	9.98	15.92	145	1
crafty	110.61	51.59	439	21
eon-cook	27.87	14.36	106	7
gap	8.08	10.28	167	1
gzip-graphic	4.62	2.88	14	1
mcf	1.54	4.02	45	1
twolf	2.82	15.04	687	2
vortex-1	14.46	15.01	141	1
vpr-route	3.59	3.94	66	1

Table 6.4: Warm-up length for warming up 50% cache blocks in L2 cache (in 100,000 instructions).

Benchmark	Mean	Standard Deviation	Max	Min
bzip2-source	177.32	233.17	999	1
gcc-166	546.30	331.57	999	1
crafty	303.68	165.11	986	28
eon-cook	155.47	272.07	998	2
gap	136.75	62.20	788	3
gzip-graphic	837.82	245.56	999	5
mcf	15.41	73.19	810	1
twolf	34.76	22.38	920	8
vortex-1	208.94	84.66	874	5
vpr-route	52.69	35.47	232	2

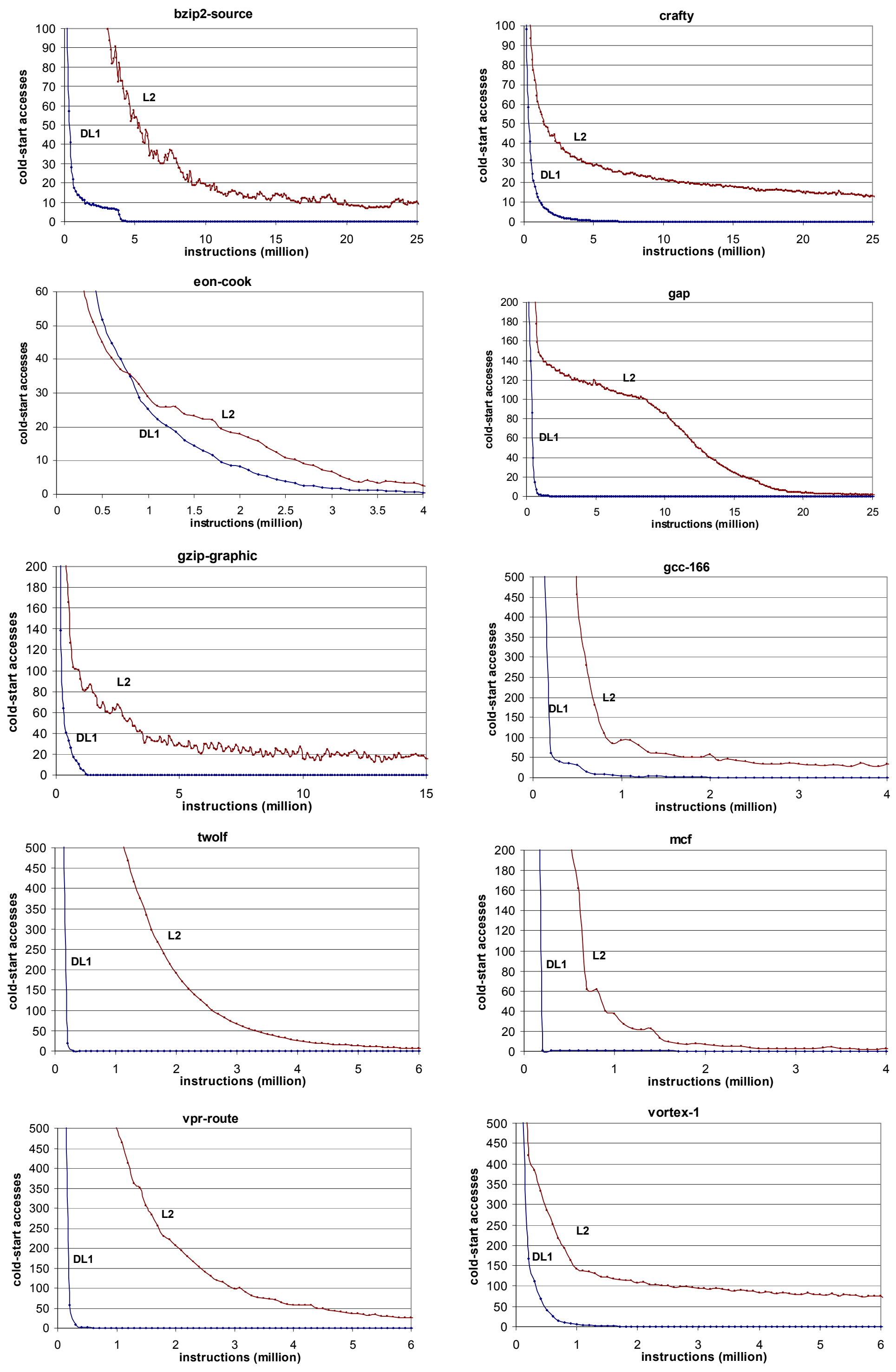

Figure 6.1: Number of cold-start cache accesses per 100,000 instructions for level 1 data cache and level 2 cache.

6.3.2 Comparison with Prior Techniques

In this section, SMA is compared with the two most recently proposed warm-up techniques, MRRL and BLRL. Both warm-up length and the accuracy in CPI are compared. In the experiment, a sampling unit size of 1 million instructions is chosen. This sampling unit size was used in the MRRL paper [29], and in Variance SimPoint [61]. In this section, each benchmark execution is divided into segments of 200 million instructions. A segment size of 200 million instructions is used instead of 100 million instructions in the previous section to give a larger gap between sampling units for more accurate profiling in MRRL and BLRL. One sampling unit is chosen from each segment.

In SMA the sampling units are not previously determined but rather depend on the cache warm-up process. Once the cache is deemed warmed up enough, the simulator executes 4,000 instructions in cycle accurate mode to warm up the pipeline as suggested by Wunderlich, *et al.* [81] and then the CPI of the 1 million instruction sampling unit is measured. As discussed in the previous section, the L2 cache may not be completely warmed up with a reasonable number of instructions so complete warm-up cannot be used as the only criterion for the caches. Therefore, the following simple heuristics are employed to judge whether the cache is warmed up. At the end of each interval, the average number of cold-start accesses for the last N intervals is calculated. If the average number of cold-start references falls below a threshold T, it is assumed that the cache is warmed up enough and the functional warm-up can be ended. Because this method requires warm-up of at least N intervals, to take advantage of segments that reach complete warm-up quickly, the number of cache blocks in the cold-start state in the cache is also monitored. The functional warm-up also ends as soon as the cold-start state blocks drop to zero. For the L1 data cache, N=20 and T=10 are used. For the L1 instruction cache, N=10 and T=1 are used. For the L2 cache, N=20 and T=15 are chosen.

For MRRL and BLRL, the sampling units are chosen to be the same as those in SMA. 4,000 instructions are also simulated in the cycle-accurate mode before each sampling units to warm up the pipeline. The profiler for MRRL was downloaded from its authors' website [27].

Although not implemented in the current simulator, I hope to further improve simulation speed by distributed simulation. When sampling units are distributed to different machines, the end state of one sampling unit cannot be used as the beginning state of another sampling unit. Therefore, in the experiment caches are cleared before warming up each sampling unit as proposed by Nguyen, *et al* [55].

The final error in CPI in sampled simulation comes from two sources: the sampling error per se and the warm-up error. To fairly compare different warm-up techniques, only the warm-up error should be measured, so an additional simulation with full cache warm-up is run. In this simulation the caches are always simulated between the sampling units as in SMARTS. The sampling units and the cycle-accurate warm-up are the same in all of the simulations. Therefore, the difference between the CPI of a warm-up technique and the CPI of full cache warm-up is the warm-up error.

The heuristics in SMA rely on the warm-up history to predict whether the cache is warmed up enough in the next sampling unit, so SMA may mispredict and end functional warm-up prematurely. Figure 6.2 shows the average warm-up error for the three warm up schemes. The error for SMA is only about 0.2%, so it is very accurate and rarely mispredicts.

For MRRL, *p*-value of 99.9%, the default value suggested by its inventors [29], is used. For BLRL, the *p*-value of 90% is chosen. Both methods are accurate, exhibiting an average error of 0.4% and 0.3%. However, SMA clearly shows the advantage as can be seen from Figure 6.3. The SMA technique on average requires only 1/3 of the warm-up length of MRRL or 1/2 of the warm-up length of BLRL yet it achieves an error that is smaller than the other two techniques. Because SMA is better in both warm-up length and accuracy, changing the *p*-value for all benchmarks will not affect the overall conclusion.

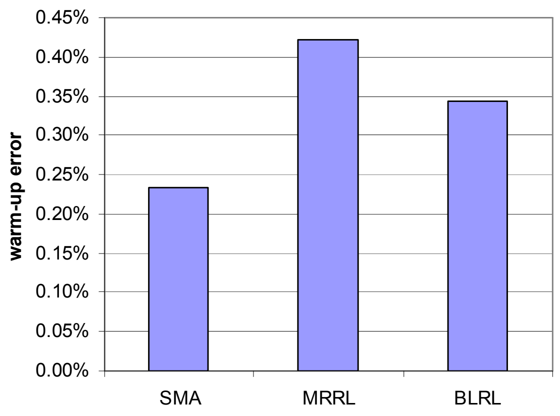

Figure 6.2: Average warm-up error of proposed SMA in comparison to other previous warm-up schemes.

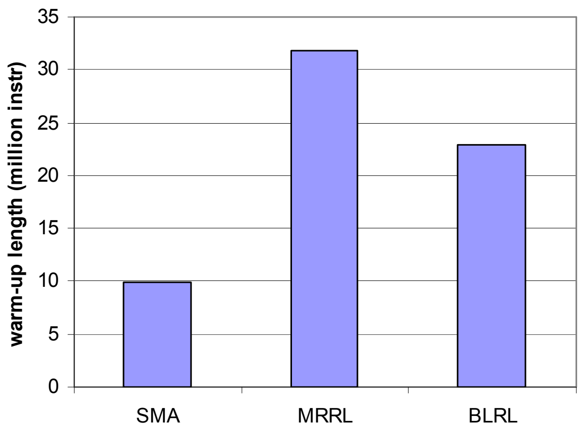

Figure 6.3: Average warm-up length per sampling unit for proposed SMA and other previous warm-up schemes.

Table 6.5 shows the detailed data for each benchmark. BLRL and MRRL do perform better for some benchmarks. This indicates that using different *p*-values for different benchmarks may improve the result of MRRL and BLRL, but asking the user to fine-tune the *p*-value for each benchmark and different processor configuration is not practical.

Table 6.5: Comparison of SMA with MRRL and BLRL.

Benchmark	# of sampling units	Avg warm-up length per sampling unit (million instructions)			Error in CPI		
		SMA	MRRL	BLRL	SMA	MRRL	BLRL
bzip2-source	545	8.7	100.2	78.6	0.1%	0.01%	0.05%
gcc-166	235	8.9	7.1	12.4	0.2%	1%	0.5%
crafty	960	13.2	3.7	14.9	1%	2%	1%
eon-cook	403	5.3	6.7	3.2	0.04%	0.3%	0.3%
gap	1346	14.4	11.6	12.1	0.04%	1%	0.1%
gzip-graphic	519	9.3	7.9	5.2	0.5%	0.2%	0.09%
mcf	310	2.7	34.7	7.6	0.004%	0.02%	0.03%
twolf	1733	6.0	15.3	5.4	0.2%	0.02%	0.3%
vortex-1	595	23.2	39.1	34.9	0.1%	0.2%	1%
vpr-route	421	7.6	91.9	55.4	0.03%	0.01%	0.02%
Average	707	9.9	31.8	23.0	0.2%	0.4%	0.3%

6.4.3 Adaptivity to Cache Configuration

Unlike previous methods, SMA adapts to the cache configuration being simulated. The previous section focuses on how SMA performs with a cache size common to workstations. To evaluate its adaptivity, in this section a small cache configuration typical in an embedded processor is also simulated. Table 6.6 shows the cache configurations used in the experiment. The small cache configuration is modeled after Intel XScale PXA255 embedded processor. Although SPEC INT is not the best benchmark suite for embedded processors, it allows comparison with the warm-up length for the large cache configuration, which is the same as used in the previous experiment.

Because MRRL or BLRL does not need to be profiled in this experiment, a segment size of 100 million instructions is used to increase the sample size. Using the same warm-up heuristic parameters as in the previous section, the average warm-up length per sampling unit for different benchmarks is shown in Figure 6.4. The first bar and the second bar for each benchmark show the warm-up length for the large cache configuration and the small cache configuration respectively. It is clear that SMA adapts well to the cache configuration. For the small caches the warm-up length is on average only 1/6 of that required by the large caches.

Neither MRRL nor BLRL adapts to the cache configuration. Using the warm-up length for MRRL or BLRL in the previous section for the small caches will result in 15~20X larger overhead than SMA. The only way to reduce warm-up length for the two techniques is to reduce the *p*-value. However, to come up with a good *p*-value for each configuration by experiment is highly impractical and defeats the goal of reducing simulation time.

Table 6.6: Configuration for caches.

	Cache	**Cache Size (bytes)**	**Block Size (bytes)**	**Associ-ativity**	**# of Sets**	**Replacement Policy**
Small cache config	L1 Data	32K	32	32	32	LRU
	L1 Instr	32K	32	32	32	LRU
	L2	None				
Large cache config	L1 Data	32K	32	2	512	LRU
	L1 Instr	32K	32	2	512	LRU
	L2	1M	64	4	4096	LRU

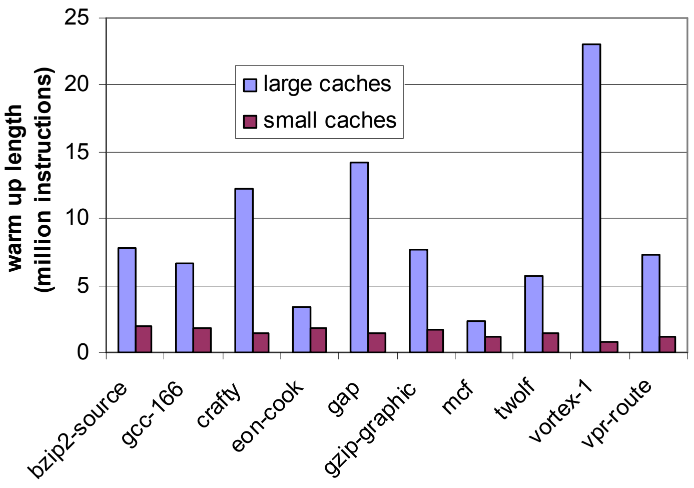

Figure 6.4: Average warm-up length per sampling unit for different cache configurations

SMA reduces warm-up overhead for small caches, but it should not compromise the accuracy of warm-up. Inadequate warm-up will cause overestimation of cache misses and eventually lead to error in CPI. The absolute error in the number of data cache misses per 1 million instructions (i.e. one sampling unit) compared with full cache warm-up is calculated. Table 6.7 shows the error averaged over all sampling units. For many benchmarks there is no error in data cache misses. For others (ie. *crafty*, *gap* and *twolf*) a small error occurs in only several sampling units. And when averaged over thousands of sampling units, the error becomes extremely small. Therefore, SMA does not lose accuracy when adapting to the small caches.

Table 6.7: Average absolute error in the number of data cache misses per 1 million instructions.

Benchmark	**Error**
bzip2-source	0
gcc-166	0
crafty	0.01307
eon-cook	0
gap	0.00670
gzip-graphic	0
mcf	0
twolf	0.05233
vortex-1	0
vpr-route	0

6.4 SUMMARY

In this chapter the warm-up process of microprocessor caches is studied in the context of sampled simulation. While sampling can greatly reduce simulation time, effective sampling requires efficient and accurate warm-up of microarchitectural structures. It is found that the warm-up process varies widely for different benchmarks, for different portions in the same benchmark execution, and for different cache configurations. Based on this observation, a self-monitored adaptive (SMA) cache warm-up scheme is proposed. The simulator monitors the cache warm-up process and decides when the warm-up is sufficient based on simple heuristics. The experiments show that SMA is accurate, exhibiting an average warm-up error of about 0.2%. SMA not only offers superior overall accuracy but also reduces the warm-up length to 1/2 ~ 1/3 of two recently proposed methods. Unlike previous methods, SMA is adaptive to the cache configuration so it can reduce warm-up overhead by an order of magnitude when simulating small caches. Because SMA continues to monitor the cache accesses during cycle accurate simulation, the user can get the number of cold-start cache accesses in each sampling unit as an indicator of the accuracy of the warm-up.

SMA also looks promising for warming up other microarchitectural structures such as the branch predictor and the value predictor. Both of them share the same property with caches that once an element is warmed up, it never goes back to cold-start state again, so they are also candidate for SMA. Unlike caches, one access to a branch table element is not sufficient to put it into a known state, so designing accurate warm-up method by tracking reuse latency as in MRRL or BLRL is not easy, but monitoring the warm-up process with Vengroff, *et al.*'s deterministic finite automaton [78] can be much simpler.

Chapter 7. Locality Based On-Line Trace Compression

7.1 INTRODUCTION

Simulation is one of the most important techniques that computer architecture researchers use to understand the behavior of complex systems and to evaluate new microarchitectural enhancements. Currently, most research in this area uses either execution-driven simulation or trace-driven simulation. Trace-driven simulation remains an important technique, especially for studying complex commercial applications, because it is usually a heroic task to set up and tune execution-driven simulators to simulate runs of commercial server benchmarks (e.g., TPC-W) while making sure that the execution delivers the best performance and meets the run rules at the same time. Traces, on the other hand, can be collected once by server performance experts, and then shared with system designers relatively easily. Moreover, if only part of the system (e.g., memory hierarchy) is to be studied, execution-driven full-system simulation is often not necessary and tends to be slower than trace-driven simulation.

One of the major problems of tracing is the high cost of storing the traces. Modern benchmarks from both scientific and commercial workloads largely resemble real applications. They often run for minutes on even GHz machines, resulting in huge trace files even after sampling.

Trace compression reduces the size of the trace file by filtering out the redundancy in the trace while retaining all the information in the original trace. Researchers have long recognized that there is abundant redundancy in program address streams [25]. The first order Markov Entropy is about 1-2 bits per address [5]. Yet the encoding based purely on coding theory is too complex and computationally expensive to implement [5]. General-purpose compression algorithms like LZW [79] (used in Unix *compress*), or LZ77 [82] (used in *gzip*) can certainly be applied to reduce the trace file size and achieve good compression ratio. Trace compression techniques, on the other hand, usually offer better compression by taking advantage of the redundancy specific to the traces. Most of the trace compression techniques work together with general-purpose compression algorithms.

Different applications require different types of traces. As a result, different compression techniques have been proposed to compress these traces. HAFT [9] effectively compresses the trace of heap allocation events (e.g., malloc, free, etc.) for studying dynamic memory allocation performance. Hamou-Lhadi and Lethbridge [16] developed a comprehension-driven compression framework that compresses the traces of procedure calls. This chapter focuses on compressing program execution traces for processor simulation. Some of these techniques rely heavily on analysis of the program's control flow information [15][21][38][63]. Although they often give the best compression ratio, they usually need to either parse the trace multiple times requiring large intermediate storage, or they need to instrument the source code or binary code of the benchmark program, a complex process often limited by the availability of tools. Instrumenting the program for tracing a full system with multiple processes and operating system activity is extremely hard. This chapter focuses on on-line (one-pass without intermediate storage) trace compression techniques without code instrumentation. Similar techniques, such as Mache, PDATS/PDI, SBC, and VPC, are discussed in the next section.

This chapter presents a trace compression scheme, the Locality Based Trace Compression (LBTC), which is suited for on-line compression of full system traces. Previous methods usually handle address only traces (Mache, PDATS) and possibly instruction words (PDI). The LBTC scheme accommodates other attributes and events in addition to address streams and the instructions themselves. Therefore, it is good for memory hierarchy simulators as well as detailed trace driven microarchitecture simulators. The compression method is based on the same spatial and temporal locality principle that has been successfully exploited by microprocessor hardware caches for decades. The resultant compression ratio is about 2X better than the PDI format.

7.2 Related Work

Previous trace compression schemes such as Mache [66] and PDATS/PDI [32] make use of the small offsets between addresses of successive memory references. The Mache scheme records addresses of three types of memory references: instruction fetch, data read, and data write. A 2-bit *label* is used to differentiate the type of reference. Three variables, initialized to zero, contain the last address in the trace for each label. Upon a new memory reference (i.e., a label, address pair) the offset is computed between this address and the last address for this label. If the offset is smaller than a given threshold, it is emitted to the trace file with the label; otherwise the full address is emitted. Because of spatial locality, the offset usually requires fewer bits than the 32-bit address. An example scheme [66] packs the data into a 16-bit word with 2 bits reserved for the label and 14 bits for the signed address offset, implying a threshold of 2^13=8192. Finally, the processed trace is passed to a general-purpose one-pass compression scheme such as the Unix *compress* utility. This seemingly simple compression scheme is very effective: Mache (with *compress*) almost always creates a file at least ten times smaller than the original trace file and over three times smaller than that produced by *compress* alone.

The PDATS scheme [32] records the memory reference address and an optional timestamp indicating when the reference occurred. A trace record in PDATS format consists of a header byte, a repetition count byte (0-1 byte), an address offset (0-4 bytes) and an optional timestamp offset. Like Mache, PDATS also takes advantage of spatial locality by encoding the offset of the addresses and timestamps, but it extends the Mache scheme in several ways. Firstly, PDATS allows variable length address offsets (1, 2 or 4 bytes). The offset is no longer limited to 14 bits. If its absolute value is less than 128, the offset will occupy only 1 byte. Secondly, PDATS takes advantage of "sequentiality." Sequentiality is an extreme form of spatial locality in which references in a stream progress monotonically through contiguous memory locations. For example, the addresses in a sequential stream of instructions from a processor with 32-bit wide instructions show a constant stride of 4. It is observed that over 90% of instruction fetches are from sequential locations. Therefore, this special offset of 4 can be encoded in the trace record header with much fewer bits instead of occupying a whole offset byte. Finally, nearly half of all the references examined are from the same stream and have the same offset as the immediately preceding reference. When repetition of the offset occurs, a repeat flag is set. The byte following the header byte specifies the number of times that this offset is repeated contiguously in the original trace. As a result, PDATS improves compression by 2X over Mache before *compress* is applied and shows 25% improvement after *compress* is applied.

The Mache and PDATS traces contain only the address information; they are suited for memory hierarchy simulation. When users need to evaluate processor designs, the trace must contain several attributes (e.g., instruction word) in addition to memory addresses. The PDI format [32] augments PDATS by including the instruction words. In PDI, addresses are compressed using a subset of the PDATS techniques, while the instruction words are compressed using a dictionary-based approach. It is observed that the 256 most frequent instruction words account for 56% to 99.9% of the instructions executed with a median of 86%. To compress the instruction words, a dictionary of the

256 most frequent instruction words, which requires only a 1-byte index to access, is stored at the beginning of the trace file. Each occurrence of these instruction words in the trace is encoded as a 1-byte index to the dictionary. On a MIPS R3000 machine, the compressed SPEC CPU92 traces in PDI format with instruction words are only 33% larger than traces in PDATS format (without instruction words).

Most recently, two new on-line trace compression techniques have been proposed. Like PDI, the Stream Based Compression (SBC) [54] also handles instruction words. It compresses the instruction sequence and data address sequence separately. The data address sequence is compressed with offset encoding similar to PDATS. The instructions are divided into "streams." A stream is a sequential run of instructions from the target of a taken branch to the first taken branch in sequence. Because the program often repeatedly executes the same stream, each stream is identified and stored in a stream table. Indexes to the stream table, instead of the streams themselves, are stored in the compressed trace file, resulting in 2 to 5 orders of magnitude reduction in Dinero+ traces[17]. Because the whole stream table is loaded into memory during decompression, its memory usage is not bounded. The memory required by SBC decompression will be approximately proportional to the instruction footprint of the program being traced.

Another trace compression technique is the value prediction based compression scheme (VPC) proposed by Burtscher and Jeeradit [7]. Many value prediction schemes have been proposed in processor architecture research for improving the performance of the microprocessor whereas Burtscher and Jeeradit used value prediction to compress traces. A hybrid value predictor consisting of several sub-predictors is created in the compression program. The predicted value is compared with the true value in the original trace. If the two values are equal, then only a bit is stored in the compressed trace indicating a correct prediction. Otherwise, the true value is stored in the compressed trace. The same predictor is also used in the decompression program, which replaces the "correct prediction" flag bit with the predicted value to get the original trace. VPC can compress load value traces by a factor of about 34 on average. The memory usage is bounded by the size of the value predictor. Because of the complexity of the value predictor, the decompression is slow compared to other methods.

7.3 Locality Based Trace Compression

This section presents the proposed Locality Based Trace Compression (LBTC) method that is suitable for on-line compression of full-system traces [50]. It compresses the trace on-the-fly without complicated analysis. It is intended to handle both CISC (variable instruction length) and RISC instruction sets efficiently. Trace files typically consist of trace records, each of which corresponds to a memory reference (data read, data write, or instruction fetch), an exception, or other event (e.g., cache flush). A trace record in LBTC captures not only the address of the memory reference but also other system information that is important for accurate memory hierarchy simulation (e.g., whether the memory location is cacheable) and accurate processor microarchitecture simulation (e.g., the instruction word). The different types of information recorded in a trace record are called *attributes*. Structurally, a trace record consists of a header and some attribute fields following it. The header tells the type of the trace record and specifies the attribute fields (e.g., whether a field exists, how long it is). Table 7.1 gives a list of these attributes. Different applications require different attributes to be recorded in the trace. After exploring the locality in the trace, each user can devise the optimal format for his or her application. Exceptions and other events are also captured in the trace file but they occur so rarely compared to memory references that their coding is of no importance to the size of the trace. Their trace records are not discussed hereafter.

Table 7.1: Example attributes recorded in a memory reference trace record

Memory reference type	Example attributes recorded in trace record
Instruction fetch	Physical address, virtual address, instruction word, mode (user/kernel), memory type (write back, uncacheable, etc.) CR3 register.
Data access	Physical address, virtual address, read/write, mode (user/kernel), size, memory type (write back, uncacheable, etc.), access type (prefetching, page table, etc.), initiating device, CR3 register.

The proposed LBTC scheme employs two techniques. One technique, offset encoding, is inherited from Mache and PDATS formats, which takes advantage of the spatial locality in memory references. The spatial locality is the property that the next address accessed will probably be very close to the last address accessed. As a result, recording the small offset between the addresses of successive references requires fewer bits than recording the full addresses in the trace file. The encoding of the instruction physical address is presented here as an example. First, the offset between the addresses of the current instruction and the last instruction is computed. A 2-bit field (`pa_code`) is allocated in the trace record header to indicate the length of the offset (see Table 7.2). Then the offset is stored after the header in 0, 1, 2 or 4 bytes. Please note that when the address of the current instruction is contiguous to the last instruction (`pa_code`=00), the current address can be calculated by adding the length of the last instruction word to the last address. Thus no extra offset byte is needed. A difference between PDATS/PDI and LBTC is the way sequentiality is exploited. PDATS/PDI format encodes an offset of 4 as 00. It works well for MIPS instructions, which have a fixed length of 4 bytes. But for x86, the instruction length ranges from 1 to 15 bytes. The `pa_code`=00 allows LBTC to encode variable length as well as fixed length instructions efficiently because contiguous instructions are often executed unless interrupted by a taken branch. Virtual addresses are treated in a similar way. In most cases, the current reference is in the same page as the last reference. Only one bit is needed to flag this situation where no extra byte for the offset is needed. In other cases, the offset of the page number is stored (1 to 3 bytes). A 4-byte offset is needed only in extreme cases.

Table 7.2: LBTC trace record header bits for instruction.

pa_code	**offset length (bytes)**	**description**
00	0	The instruction is contiguous to last instruction (i.e. its address is the address of the last instruction plus the length of last instruction)
01	1	The offset is between –128 and 127
10	2	The offset is between –32768 and 32767
11	4	The offset can only be expressed in 4 bytes

The other technique in LBTC is based on the "static" property of most of the attributes, and the well-known temporal locality of memory references. Because the trace contains instruction words and other attributes in addition to physical addresses, the attribute information takes more than half of the space. Encoding the most frequent 256 instruction words in 1 byte as in the PDI format is not effective enough, as shown in Section 7.4.2. Fortunately, most of the attributes are "static". They do not change frequently from one dynamic access to another, if the references are to the same memory addresses or initiated by the same instruction. For example, after a non-self-modifying benchmark program is loaded, the (static) instruction at a specific address will remain the same until the program terminates (or is swapped out onto disk) and a new module is loaded at the same memory area. Therefore, the next (dynamic) instruction fetched from the same address will probably have the same instruction word. This holds true for many other attributes such as virtual addresses, memory type, etc. For all the "static" attributes, the temporal locality can be effectively employed for compression. Memory references from programs are known to show abundant temporal locality: if a memory address is accessed, it will probably be accessed again in the very near future. Therefore, a small direct mapped cache structure, called a *compression cache*, is emulated in software, to keep the recently seen memory references. If the next memory reference hits in the cache, all the “static” attributes can be retrieved from it without being stored in the trace file. The compression cache is indexed with the physical address of instruction fetches.

In LBTC, no separate data cache is created for data references. Instead, the data reference attributes are attached to the instruction before it. A data reference that appears after an instruction is probably initiated by that instruction, but there is also a slight possibility that the data reference has been caused by DMA operations or by TLB misses. If the data reference attached to a dynamic instruction is the same as the data reference to the last execution of the same static instruction, and the instruction is found in the cache, then it is called a *data reference hit*. The data reference record is then retrieved from the compression cache with the instruction physical address as the index. In this way, on a data record hit, even the data address can be omitted in the trace. Only one bit in the header is needed to indicate that the data can be retrieved from the cache during decompression. It works because many instructions will access the same memory address as its last execution. Another approach is to create a separate data cache indexed by the physical address of data references. The trade-off is briefly discussed in Section 7.4.1. The compression cache in the implementation contains only 32K entries, thus it does not tax the virtual memory system. It is also very fast because one access involves only one bit-wise *and* operation and one indexing operation. Set associative or fully associative data structures may improve the compression ratio, but require much more time to access. The working of the compression cache is analogous to that of a hardware cache.

The C-like pseudo code in Figure 7.1 illustrates the steps in LBTC compression. The direct mapped compression cache structure is not part of a trace file, but is created on-the-fly and used in the program. Only the cache parameter (i.e., number of entries) needs to be passed to the compression and the decompression program. The tracer is a utility such as the trace module in Simics [52], from which the original uncompressed trace records are obtained. A `cache_entry_t` structure contains the trace record of an instruction fetch and the data memory reference records following it. The `emit` statement in the figure writes the trace information to the compressed trace file.

Table 7.3 gives an example to illustrate the steps used to produce a compressed trace. Column 1 assigns a number to each trace record. Column 2 gives an ID to each static instruction. Column 3 numbers the dynamic data references. Column 4 shows the physical addresses of the memory references. The next 4 columns are the contents in the compressed trace. Column 5 is a bit showing the type of the trace record (I for instruction, D for data). The algorithm converts the physical addresses to address offsets, which are stored in the trace file in two's complement using the minimum number of bytes required to hold the offset. As in Mache and PDATS/PDI format, the address offsets are only calculated between successive references of the same type (data access or instruction fetch) but data accesses are not further split into sub-types. The first address of each type is simply reproduced in the compressed file. The offsets stored in the trace file are in column 6. Column 7 is the 1-bit field in every trace record header indicating whether the reference can be retrieved from the compression cache (M for cache miss, H for cache hit). In this example, a direct-mapped cache with 4 entries is assumed. The physical address of the instruction is used to calculate the index to the cache, whose operations are shown in the last column. Before each instruction is put into the cache, the cache is probed to see if the instruction is already there. If the instruction is already in the cache (a cache hit), only the address offset is stored in the trace file with a flag indicating a cache hit. On a cache miss, all the attributes are stored. Each data reference is associated with the instruction before it, thus in the same cache entry as the instruction.

```
uncompressed_trace_record_t tc;
cache_entry_t *current_entry=NULL;
cache_entry_t *entry_in_cache=NULL;

create and initialize compression cache structure;
while(tracer has more trace records){
  get a record from tracer and store it in tc;
  emit tc.type;
  if(tc.type==INSTRUCTION){ // trace record for instruction fetch
    emit offset of tc.physical_address;
    search cache using tc.physical_address as index and
      assign the cache entry to entry_in_cache;
    if(entry_in_cache!=NULL
       && entry_in_cache->instruction_record==tc){
      // found in cache
      emit hit flag;
    }else{ // not found in cache
      emit all attributes;
    }
    allocate and initialize current_entry;
    add tc to *current_entry;
    put current_entry into compression cache;
  }else{ // trace record for data reference
    if(entry_in_cache!=NULL &&
       (the corresponding entry_in_cache is the same as tc)){
      // data reference found in cache
      emit hit flag;
    }else{ // this data reference not found in cache
      emit offset of tc.physical_address;
      emit all attributes;
    }
    add tc to *current_entry; // put tc into the cache
  }
}
```

Figure 7.1: Pseudo code illustrating the LBTC algorithm.

Table 7.3: Example of trace compression using LBTC.

Trace record	Static instruc-tion numbe r	Data refer-ence number	Physical address (HEX)	Contents of compressed trace				Cache operation
				Type	Offset (HEX)	Hit/Miss flag	Other attributes	
#1	I1		ae05	I	ae05	M	Instr word, etc	Miss, put in I1
#2	I2		ae07	I	2	M	Instr word, etc	Miss, put in I2
#3		D1	cfb8	D	cfb8	M	Virtual addr, memory type, etc	put in D1
#4	I3		ae00	I	-7	M	Instr word, etc	Miss, put in I3
#5		D2	cfe8	D	30	M	Virtual addr, memory type, etc	put in D2 (Table b)
#6	I1		ae05	I	5	H	--	Hit
#7	I2		ae07	I	2	H	--	Hit
#8		D3	cfb8	D	-30	H	--	Hit (Table c)
#9	I4		ae08	I	1	M	Instr word, etc	Miss, put in I4, replace I3
#10		D4	cfa0	D	-18	M	Virtual addr, memory type, etc	put in D4
#11		D5	cff0	D	50	M	Virtual addr, memory type, etc	put in D5
#12	I5		ae06	I	-2	M	Instr word, etc	Miss, put in I5 (Table d)
#13	I4		ae08	I	2	H	--	Hit
#14		D6	cfa0	D	-50	H	--	Hit
#15		D7	cff4	D	54	M	Virtual addr, memory type, etc	Miss, put in D7, replace D5 (Table e)

Table 7.4: Compression cache content after record #5

Index	Content
0	I3, D2
1	I1
2	
3	I2, D1

Table 7.5: Compression cache content after record #8

Index	Content
0	I3, D2
1	I1
2	
3	I2, D3

Table 7.6: Compression cache content after record #12

Index	Content
0	I4, D4, D5
1	I1
2	I5
3	I2, D3

Table 7.7: Compression cache content after record #15

Index	Content
0	I4, D6, D7
1	I1
2	I7
3	I2, D1

At the time of compression, if the physical address of an instruction is found in the cache, all attributes of the instruction (or data) trace record will be compared with those in the cache (lines 14 and 25 in Figure 7.1). To ensure correctness, only when every attribute matches will it be considered a cache hit. Finding the same physical address in the cache does not guarantee that the same instruction is found because the operating system may have loaded a new program at the same memory area, or the program might be self-modifying. Tables 7.4 to 7.7 show the contents of the compression cache at the selected points during compression. After trace record #5, instructions I1, I2, I3 and the data references are put into the cache (Table 7.4). Trace records #6, #7 and #8 are found in the cache; the full attributes are not stored in the trace file (Table 7.5). Records #9 to #12 incur cache misses (Table 7.6) and are stored in the trace with full attributes. Please note that I4 replaced I3 just as in a regular hardware cache operation. Trace record #13 and one following data reference (D6) hit in the cache (Table 7.7), but the next data reference (D7) is a cache miss and all its attributes must be stored. After the encoding, a general-purpose compression utility such as *gzip* is applied to further reduce the file size.

7.4 Results

Traces are gathered using Simics/x86 [52], an x86 full system simulator. In the experiments, Simics simulates a Pentium II machine running Red Hat Linux 7.3. SPECint2000 is chosen to represent CPU intensive benchmarks and SPECweb99 [74] to represent commercial servers. For SPECweb, Apache 2.0 is used as the web server and mod_perl [4] is deployed to speed up some dynamic web operations. Unless otherwise noted, about one hundred million instructions are gathered for each benchmark. Approximately two billion instructions are skipped for every SPECint2000 benchmark before tracing.

7.4.1 Compression Ratio

Many general-purpose compression algorithms can be used to reduce trace file sizes. LBTC is evaluated against *gzip*, one of the most popular general-purpose compression utilities. *Gzip* implements the LZ77 algorithm [82], which strives to find the repetitive patterns within a sliding window. In the experiment the Simics trace file in its uncompressed raw binary format is gathered. To make the raw trace file size manageable, 10 million instructions, instead of 100 million instructions, are collected in this *gzip* comparison experiment. Figure 7.2 shows the size of compressed trace normalized to the size of the raw trace file. The columns for SPECint show average values of all programs. The error bars show the maximum and the minimum values. The Simics raw trace format uses union structure to accommodate the max size of different types of trace records. Therefore, each trace record in the trace file may contain unused bits. These unused bits may take on arbitrary values making an otherwise repetitive record look different and rendering *gzip* less effective. To ensure a fair comparison, these unused bits are forced to be zero. The Simics trace module is also modified to set its data value field to zero because data value of memory loads/stores are not used in LBTC. Figure 7.2 shows that *gzip* reduces the size of the trace file by an order of magnitude but LBTC alone yields better a compression ratio. LBTC wins by taking advantage of the knowledge of the structure of the trace record whereas *gzip* blindly searches for repetition. The most notable property of LBTC in Figure 7.2, which is also true for PDATS/PDI and Mache, is that it is complementary to general-purpose compression techniques. As shown in the third column for each benchmark, *gzip* can further compress LBTC format by a ratio of 4-6. On average, a trace record takes 0.357 bytes in *gzipped* LBTC files.

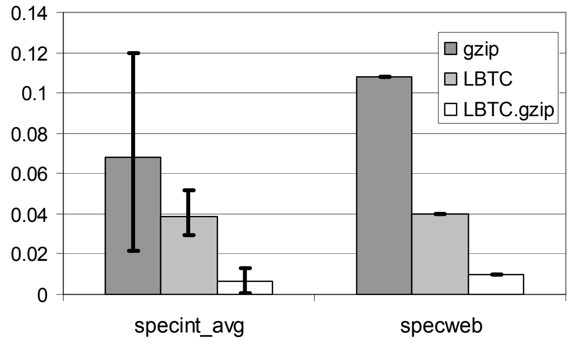

Figure 7.2: Normalized trace file size for *gzip* and LBTC.

Figures 7.3 and 7.4 compare different trace compression techniques. File sizes are normalized to the baseline compression. The baseline compression, denoted as “offset encoding” in the figure, compresses only memory reference addresses (physical and virtual) by offset encoding. It is essentially the Mache format with all the additional attributes uncompressed. Offset encoding of physical and virtual addresses is also used in PDI and LBTC in the experiment. The PDI format does further compression by encoding the most frequent 256 instruction words in 1 byte. When the top 256 instructions are obtained from the trace itself, it is denoted as “specific”. This approach cannot be used on-line because it requires two passes. The first pass scans the trace to identify the 256 most common instruction words. The second pass does the encoding. An alternative approach uses a “generic” dictionary that is selected for x86 after examination of a collection of traces. In the experiment, the generic dictionary is obtained by calculating the top 256 instructions for all benchmark programs. This will usually yield less compression but permits on-line compression. Figure 7.4 shows the same results after files are further compressed by *gzip*. As shown in Figure 7.3, the generic dictionary offers little compression for SPECweb. Even within SPECint groups, some benchmarks are so different that the generic dictionary is of little use. LBTC offers 2.5X better compression over PDI (generic) and is about 2X better after *gzip* compression is applied.

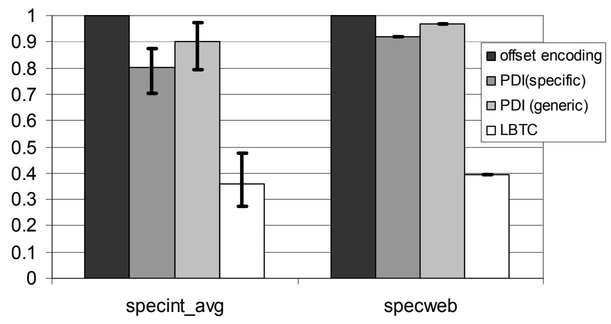

Figure 7.3: Normalized trace file size for different trace compression formats without *gzip*.

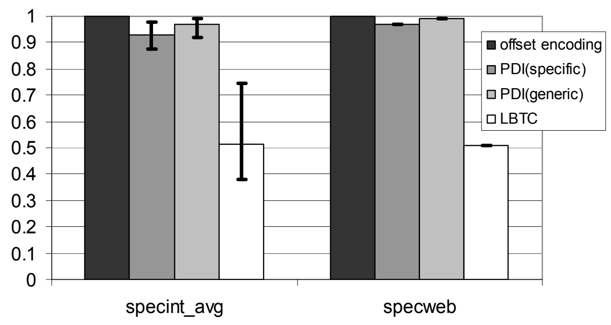

Figure 7.4: Normalized trace file size for different trace compression formats with *gzip*.

It can be seen that the PDI format is not very effective in the experiment. The PDI format is most effective if the following three conditions are met:

1. The instruction word constitutes the major part in a trace record besides the address information.
2. The 256 most frequent instruction words account for most dynamic instruction words in the trace.
3. The 256 most frequent instruction words are long so that replacing them with 1-byte indices provides good saving.

As for condition 1, the virtual address, the memory type, and other information are also traced besides the instruction word and the physical address. PDI compresses only the instruction word and thus is not effective for other information. On a cache hit, LBTC retrieves all information from the cache. Therefore, storing extra information is not a problem as long as this information does not change frequently from one execution of the same instruction to the other.

Condition 2 is met for most SPECint programs. As shown in Table 7.8, the top 256 instructions account for an average of 90% of all dynamic instructions. Yet instructions used in commercial servers like SPECweb are more diversified. The top 256 instructions only cover 56% of the total instructions. Unlike MIPS, for which PDI was first designed, x86's instruction coding is fairly compact. The average length of the instructions is about 2.9 bytes whereas for the most common 256 instructions, the average length is even smaller. Therefore, as for condition 3, replacing the instruction words with 1 byte does not provide as much compression as on MIPS, whose instruction length is 4 bytes.

Table 7.8: Statistics of the 256 most common instructions.

Benchmark		**Coverage of top 256 instrs**	**Avg length of top 256 instrs**	**Avg length of all dynamic instrs**
SPECint	Average	0.880	2.845	2.994
	Median	0.908	2.776	2.887
	Max	0.998	3.490	3.683
	Min	0.569	2.034	2.165
SPECweb		0.558	2.239	3.011

7.4.2 Statistics Supporting Compression

Offset encoding takes advantage of spatial locality by encoding the offset in fewer bytes. LBTC is similar to Mache and PDATS in this aspect and readers can obtain more details and statistics about the offset from the original reference [32]. LBTC also employs temporal locality by caching a small number of previously seen trace records. It is used to compress information other than the instruction physical address, including the instruction word, virtual address, etc., that are mostly "static" (not changing in each dynamic reference to the same memory location). The subsequent trace records can probably be retrieved from the cache, if it has been seen not long ago. Table 7.9 shows the hit ratio of the compression cache.

Table 7.9: Compression cache hit rate.

Benchmark		Instruction hit rate	Data hit rate
SPECint	Average	99.0%	51.7%
	Median	99.8%	61.5%
	Max	100.0%	79.9%
	Min	96.1%	11.0%
SPECweb		94.5%	52.8%

Even though it is a small direct mapped 32k entry cache, more than 94.5% of the instruction records can be retrieved from the cache, and about half of the data records are found in the cache. The relatively low data hit rate is due to indexing the cache with the instruction physical address. The other possible locality based approach is to use a separate data cache indexed with the address of the data reference, which would improve the cache hit ratio for data references to about 90%. This would reduce the attributes stored in the trace file, but every data reference record would require an address as the index. In the current implementation, if a data record hits in the cache, even the data address is not necessary in the trace file. Only one bit in the trace record header indicating a data hit needs to be present. Since the physical address is a significant part in the data trace record, the current implementation gives a better overall compression ratio than a separate data cache indexed by the address of the data reference. However, in other environments if the data hit ratio indexed by instruction address is low, and the data trace record contains much more than the physical address, then a separate data cache will be justified.

7.4.3 Access Time

Trace access (decompression) time is also an important metric to evaluate a trace compression method. The access time of different compression formats as well as the execution-driven tracing time is compared. Trace access time is heavily affected by disk access time. In the experiment a fast disk configuration is used. The access times were measured by reading each trace file from a SCSI RAID (level 0, two disks) attached to an 8-processor DELL PowerEdge 8450 server running Red Hat Linux 7.3. The real (elapsed) time spent in reading and converting each trace was measured using the Unix time utility.

Again, the access times to compressed trace file (not *gzipped*) are normalized to the "offset encoding" format and shown in Table 7.6. All formats perform similarly in terms of access time. This indicates that the compression cache adds very little time overhead. It is as fast as the PDI format. If the disk is slow, LBTC is expected to show relatively faster results because it has a better compression ratio and thus less time is spent reading from the disk compared to other compression methods. The access time to trace files that were further compressed by *gzip* was also measured. The files were *gunzipped* and piped into the trace decompressors. On multi-processor machine used in the experiment, the *gunzip* and trace decompressor run in parallel and the access time stays almost the same as in Table 7.10.

The last column shows the normalized time for generating the trace information in Simics. It is the overhead in execution-driven simulation. The original trace module shipped with Simics is used and the raw format tracing is turned on, which just dumps the trace records without any processing. The trace dump is redirected to `/dev/null`, so there are no disk writes. It is interesting to note that decompressing traces are faster than executing the benchmark in Simics. This is because executing a benchmark in a full system simulator is a costly process itself. Moreover, because Simics is a commercial product, to keep the source code secret, the trace module is "hooked" to the simulator core through the call back API, which causes additional large overhead.

Table 7.10: Normalized access time.

Benchmark		**Offset encoding**	**PDI (specific)**	**PDI (generic)**	**LBTC**	**Exec**
SPECint	Average	1.000	1.052	1.049	0.993	4.390
	Median	1.000	1.051	1.047	1.007	4.425
	Max	1.000	1.062	1.059	1.111	4.761
	Min	1.000	1.046	1.043	0.875	4.175
SPECweb		1.000	1.055	1.052	1.078	4.585

7.4.4 Design Space Exploration

Since the algorithm is based on the same principle of locality as hardware caches, any approach that enhances the cache hit ratio should also improve the compression ratio. For comparison, a 2M-entry direct mapped cache and an infinite fully associative cache are emulated. The compressed file sizes, after *gzip* is applied, are shown in the Table 7.11. File sizes are normalized to the 32k cache case. Table 7.11 shows that the 32K entry direct mapped cache works fairly well. The average reduction in file size of SPECint is only about 10.8% when a 2M direct mapped cache is used. Yet there are some applications, notably the SPECweb and *eon* from SPECint2000, which can benefit from larger caches. Fortunately, a larger direct mapped cache is as fast as a small cache. Thus when memory is abundant, a larger cache could be used. Moving further to an infinite fully associative cache gives less than 1% improvement. Therefore, fully associative cache, which requires a slow associative search, is not needed.

Table 7.11: Normalized file size of different compression cache configuration (with *gzip*).

Benchmark		**32k direct-mapped**	**2M direct-mapped**	**Infinite fully associative**
SPECint	Average	1.000	0.892	0.891
	Median	1.000	0.927	0.927
	Max	1.000	0.997	0.997
	Min	1.000	0.656	0.656
SPECweb		1.000	0.717	0.711

LBTC could also be combined with PDI compression. But the experiment shows that it will achieve little further compression. In addition, using PDI on-line requires creating a generic instruction word dictionary, which is a difficult task. Therefore, adding PDI to LBTC is unnecessary.

7.5 Summary

Processor memory references exhibit spatial and temporal locality. Previous research has employed spatial locality to compress address traces by encoding the offset of the consecutive reference addresses. It works well for traces that only contain addresses. In this chapter, a new technique, LBTC, is proposed to take advantage of the temporal locality by using a small data structure emulating a cache. LBTC can efficiently compress traces with more information than addresses. It is shown to improve the compression ratio by about 2X over the PDI format, which uses a dictionary to compress instruction words. The algorithm is simple, fast and can be used on-line in conjunction with general-purpose compression algorithms.

Chapter 8. Conclusions and Future Research

8.1 CONCLUSIONS

Before a new computer has been built and its real performance can be measured, simulation is the most important tool for computer architects to evaluate design tradeoffs. However, simulation time has been increasing. Software applications are becoming bigger and bigger as users demand more functionality or try to solve more difficult problems. Computer systems are becoming more and more complex as designers continue to add more transistors to achieve better performance. Therefore, simulating big benchmarks on complex computer models takes a prohibitive amount of time. On the other hand, the accuracy requirement for the simulation result has never been higher. The competition in computer industry and academic research is fierce. Computer architects often have to decide whether to incorporate an enhancement that gives a few percentage of performance improvement. Simulation experiments must be able to discern such small performance difference. As a result, reducing simulation time while maintaining high fidelity in simulation is a challenging and imperative problem.

Sampling can achieve significant simulation time reduction with good accuracy. There are many sampling techniques. No single solution is the best for all situations, just as no single processor is the best for all applications. This research has studied different sampling techniques and proposed improved techniques optimized for different user objectives and workloads. The following are the major findings and contributions to key areas in sampled processor simulation.

- **Choice of sampling unit size**

  How to choose a good sampling unit size is a basic question in sampled simulation but there has been no consensus. A large range of sampling unit sizes has been proposed and used. This research studies the question using statistical sampling theory. Under the assumption of simple random sampling and perfect warm-up, the effectiveness of a sampling unit size depends on the sign of the intracluster correlation coefficient. It is observed that in nearly all cases using small sampling units produces more accurate result given the same simulation budget. A more important contribution is that the study discerns the inherent temporal locality in the benchmark that underlies the observation. It is found that, although still popularly used, simulating one very large chunk of instructions is not an efficient way to improve accuracy [46].

- **Simulating commercial workloads**

  Commercial workloads such as database servers are very important in the business world. Simulations of those benchmarks are harder to set up and take longer to run than SPEC CPU. However, simulation time reduction techniques for commercial workloads have not been adequately studied. This research studies two such techniques, simple random sampling (with small unit size) and representative sampling, for the application of OLTP workloads. Although OLTP workloads do not exhibit long, continuous phases, representative sampling is still applicable. But its effectiveness highly depends on the chunk size. Users should carefully choose the chunk size to get accurate results. Random sampling with small unit size like SMARTS is also good at reducing simulation time for OLTP workloads. However, OLTP workloads are different from SPEC CPU programs. OLTP workloads are composed of randomly generated sequence of relatively short database transactions, so at a large interval, execution of an OLTP workload is stationary. Based on this property, a dynamic stopping rule is proposed. The simulator monitors the current confidence interval as the simulation proceeds. It stops after the confidence interval has met the user's target accuracy requirement. The dynamic stopping rule eliminates the second round of simulation that is usually required in a popular prior technique to meet the user's target accuracy. It improves the usability and reduces the total simulation time.

- **Measuring relative performance improvement**

  The objective of most simulations is to find out the performance benefit of some microarchitecture enhancement. In these simulations, users care more about the accuracy of the speedup than the accuracy of CPI. Nonetheless, previous research has been focusing solely on CPI. By employing the ratio estimator from statistical sampling theory, this research presents an efficient sampling technique to measure speedup and to quantify its error. Because the executions of the same benchmark on two similar configurations are highly correlated, to achieve a given relative error limit for speedup, it is not necessary to estimate CPI to the same accuracy. In the experiment, estimating speedup requires only about 1/9 of the instructions needed for estimating CPI for the same relative error limit. Therefore using the ratio estimator to evaluate speedup is much more cost-effective and offers great potential for reducing simulation time [49].

- **Adaptive warm-up**

  In order to achieve accurate sampling results, microarchitectural structures must be adequately warmed up before each measurement. Warm-up is an important issue in sampled processor simulation because it is critical to the accuracy of the result and it also incurs simulation overhead. Previous cache warm-up methods do not take into account the cache configuration being simulated, an important factor in the warm-up process. In this dissertation, a new technique for warming up microprocessor caches is proposed. The simulator monitors the warm-up process of the caches and decides when the caches are warmed up based on simple heuristics. The Self-Monitored Adaptive (SMA) warm-up technique on average exhibits only 0.2% warm-up error in CPI. SMA achieves smaller warm-up error with only 1/2~1/3 of the warm-up length of previous methods. In addition, it is adaptive to the cache configuration simulated. For simulating small caches, the SMA technique can reduce the warm-up overhead by an order of magnitude compared to previous techniques. Finally, SMA gives the user some indicator of warm-up error at the end of the cycle-accurate simulation that helps the user to gauge the accuracy of the warm-up [47][48].

- **Trace compression**

  In trace-driven simulation, sampled traces have to be stored. This dissertation also investigated trace compression to reduce the cost of storage. Although generic compression method like *gzip* can be used, compression techniques designed specifically for traces give higher compression ratio. Previous trace compression schemes such as Mache and PDATS/PDI take advantage of spatial locality to compress memory reference addresses. This research presents the Locality Based Trace Compression (LBTC) method, which employs both spatial locality and temporal locality in program memory references. It efficiently compresses not only the address but also other attributes associated with each memory reference. In addition, LBTC is designed to be simple and on-the-fly. If traces with addresses and other attributes are compressed by LBTC, the compression ratio is better by a factor of 2 over compression by PDI [50].

8.2 DIRECTIONS FOR FUTURE RESEARCH

- **Simulation for multiprocessor systems**

  As Moore's law dictates, more and more transistors are available now, but it has been increasingly difficult to use the extra transistors to improve the performance of a uniprocessor. So chip makers resort to multi-core designs. Multiprocessors have traditionally been limited to high-end systems. Since Intel and AMD have released dual-core processors for desktop, we are about to see proliferation of multiprocessor computers. Simulating multiprocessors is more difficult than uniprocessors. Although in a real system, multiple processors work in parallel to improve performance, most of the simulators simulate the processors sequentially, resulting in longer simulation time. Research on simulation time reduction techniques for multiprocessors has been limited, but study by Ekman and Stenstrom indicates that simulation of multiprocessors may have greater potential for time reduction [20]. Designing better sampling techniques for simulating multiprocessor systems is a promising research area.

- **Simulating emerging workloads**

  Different types of workloads exhibit different characteristics. Although the basic ideal of sampling still applies, taking advantage of the workload-specific characteristics often enables better sampling designs as shown in Chapter 4 for OLTP workloads. There are still important workloads such as Java servers, for which simulation time reduction techniques have not been fully studied. Emerging workloads like life science workloads are often huge and demand better time reduction techniques.

- **Measuring power, reliability, etc**

  This dissertation has been focusing on performance simulation. However, performance is not the only objective in computer design. Power has increasingly become a limiting factor. In the performance simulation, the user wants to find out the average performance. But in power simulation, the user may want to know the maximum power or the highest temperature during the execution of the benchmark. Traditional sampling is ill suited for estimating max or min values in population. Therefore, simulating for maximum power poses a challenging problem. As the wires continue to shrink, noise in chips rises and thus soft errors become a big issue in processor design. Therefore, simulation for gauging reliability will be more important in the future. More statistics are needed to handle the simulation of low probability events such as soft errors. In addition, simulations in this study may not be valid for functional validation. I/O performance was not considered in this research due to the lack of good models for I/O devices. I/O activity may not affect processor performance but it is an important factor in the system performance for many commercial workloads. How I/O simulation affects sampling design also needs further study.

Bibliography

[1] A. Agarwal, J. Hennessy and M. Horowitz, "Cache Performance Of Operating System and Multiprogramming Workloads," *ACM Transactions on Computer Systems*, vol. 6, pp. 393-431, 1988.

[2] A. Agarwal and M. Huffman, "Blocking: Exploiting Spatial Locality for Trace Compaction," *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 48- 57, 1990.

[3] A. R. Alameldeen and D. A. Wood, "Variability in Architectural Simulations of Multi-threaded Workloads," *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 8-12, 2003.

[4] Apache Software Foundation, mod_perl Home Page, http://perl.apache.org/ (accessed Dec 2003).

[5] J. C. Becker, A. Park and M. Farrens, "An Analysis of the Information Content of Address Reference Streams," *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 19-24, 1991.

[6] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Technical Report 1342*, Computer Sciences Department, University of Wisconsin-Madson, June 1997.

[7] M. Burtscher and M. Jeeradit, "Compressing Extended Program Traces Using Value Predictors," *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 159-169, September 2003.

[8] H. W. Cain, K. M. Lepak, B. A. Schwartz and M. H. Lipasti, "Precise and Accurate Processor Simulation," *Proceedings of the 5th Workshop On Computer Architecture Evaluation Using Commercial Workloads (CAECW)*, pp. 13-22, February 2002.

[9] T. Chilimbi, R. Jones and B. Zorn, "Designing a Trace Format for Heap Allocation Events," *ACM SIGPLAN Notices*, vol. 36, pp. 35-49, 2001.

[10] W. G. Cochran, *Sampling Techniques*, 3rd ed., New York: John Wiley & Sons, 1977.

[11] 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*, vol. 47, pp. 714–720, 1998.

[12] T. M. Conte, M. A. Hirsch and K. N. Menezes, "Reducing State Loss For Effective Trace Sampling of Superscalar Processors," *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, pp. 468-477, 1996.

[13] P. Crowley and J. L. Baer, "On the Use of Trace Sampling for Architectural Studies of Desktop Applications," *Proceedings of the 1999 SIGMETRICS Conference*, pp. 208-209, May 1999.

[14] S. Dasgupta, "Experiments with Random Projection," *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, pp. 143–151, 2000.

[15] L. DeRose, K. Ekanadham, J. K. Hollingsworth and S. Sbaraglia, "SIGMA: a Simulator Infrastructure to Guide Memory Analysis," *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1-13, 2002.

[16] R. Desikan, D.C. Burger and S.W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pp. 266-277, July 2001.

[17] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," http://www.cs.wisc.edu/~markhill/DineroIV/ (accessed Dec 2003).

[18] L. Eeckhout, S. Eyerman, B. Callens and K. De Bosschere, "Accurately Warmed-Up Trace Samples for the Evaluation of Cache Memories," *Proceedings of the 2003 High Performance Computing Symposium*, pp. 267-274, 2003.

[19] L. Eeckhout, Y. Luo, K. Bosschere and Lizy K. John, "BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation," *The Computer Journal*, vol. 48, pp. 451-459, 2005.

[20] M. Ekman and P. Stenstrom, "Enhancing Multiprocessor Architecture Simulation Speed Using Matched-Pair Comparison," *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 89-99, March 2005.

[21] E. N. Elnozahy, "Address Trace compression Through Loop Detection and Reduction," *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 214-215, 1999.

[22] Example Error Rates for SimPoint, http://www.cs.ucsd.edu/~calder/simpoint/error-rates.htm (accessed June 2005).

[23] J. W. C. Fu and J. H. Patel, "Trace Driven Simulation Using Sampled Traces," *Proceedings of the 27th Hawaii International Conference on System Sciences (Vol. I: Architecture)*, pp. 211–220, January 1994.

[24] S. Girbal, G. Mouchard, A. Cohen and O. Temam, "DiST: a Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 1-12, 2003.

[25] D. W. Hammerstrom and E. S. Davidson, "Information Content of CPU Memory Referencing Behavior," *Proceedings of the 4th Annual Symposium on Computer Architecture*, pp. 184-192, 1977.

[26] A. Hamou-Lhadj and T. C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces," *Proceedings of 10th International Workshop on Program Comprehension (IWPC'02)*, pp. 159-168, June 2002.

[27] J. W. Haskins, "Memory Reference Reuse Latency: Rapid Warm Up for Sampled Microarchitecture Simulation," http://www.cs.virginia.edu/~jwh6q/mrrl-web/

[28] J. W. Haskins, Jr. and K. Skadron, "Minimal Subset Evaluation: Rapid Warm-Up For Simulated Hardware State," *Proceedings of the 2001 International Conference on Computer Design*, pp. 32-39, September 2001.

[29] J. W. Haskins, Jr. and K. Skadron, "Memory Reference Reuse Latency: Accelerated Sampled Microarchitecture Simulation," *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 195-203, March 2003.

[30] V. S. Iyengar, L. H. Trevillyan and P. Bose, "Representative Traces for Processor Models with Infinite Cache," *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pp. 62-72, 1996.

[31] L. M. Jimeno-Ochoa, P. Ibez and V. Vials, "Warm Time Sampling: Fast and Accurate Simulation of Cache Memory," *Proceedings of the 22nd Euromicro International Conference*, pp. 39-44, 1996.

[32] E. Johnson, J. Ha and M. B. Zaidi, "Lossless Trace Compression," *IEEE Transactions on Computers*, vol. 50, pp. 158-173, 2001.

[33] R.E. Kass and L. Wasserman, "A Reference Bayesian Test For Nested Hypotheses And its Relationship to Schwarz Criterion," *Journal of the American Statistical Association*, vol. 90, pp. 928-934, 1995.

[34] R. E. Kessler, M. D. Hill and D. A. Wood, "A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches," *Technical Report 1048*, Univ. of Wisconsin-Madison Computer Sciences Dept., September 1991.

[35] A. KleinOsowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, vol. 1, pp. 10-13, 2002.

[36] T. Lafage and A. Seznec, “Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream,” *Proceedings of the 3rd IEEE Annual Workshop on Workload Characterization*, pp. 102-110, 2000.

[37] S. Laha, J. H. Patel and R. K. Iyer, “Accurate Low-Cost Methods For Performance Evaluation Of Cache Memory Systems,” *IEEE Transactions on Computers*, vol. 37, pp. 1325– 1335, 1988.

[38] J. R. Larus, “Efficient Program Tracing,” *Computer*, vol. 26, no. 5, pp. 52-61, May 1993.

[39] J. Lau, S. Schoenmackers and B. Calder, “Structures for Phase Classification,” *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 57-67, March 2004.

[40] G. Lauterbach, “Accelerating Architectural Simulation by Parallel Execution of Trace Samples,” *Proceedings of the 27th Hawaii International Conference on System Sciences, Volume 1: Architecture*, pp. 205-210, January 1994.

[41] M. H. Lipasti and J. P. Shen, “Exceeding the Dataflow Limit via Value Prediction,” *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 226-237, 1996.

[42] W. Liu and M. Huang, “EXPERT: Expedited Simulation Exploiting Program Behavior Repetition,” *Proceedings of the 2004 International Conference on Supercomputing (ICS’04)*, pp. 126-135, June 2004.

[43] L. Liu and J. Peir, “Cache Sampling by Sets,” *IEEE Transactions on VLSI Systems*, vol. 1, pp. 98-105, 1993.

[44] Y. Luo and L. John, “Workload Characterization of Multithreaded Java Servers,” *Proceedings of 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 128-136, 2001.

[45] Y. Luo and L. John, “Simulating Java Commercial Throughput Workload: a Case Study,” *2005 IEEE International Conference of Computer Design (ICCD)*, to appear.

[46] Y. Luo and L. K. John, “On Sampling Unit Size in Sampled Microprocessor Simulation,” *Proceedings of the 24th IEEE International Performance Computing and Communications Conference*, pp. 81-90, April 2005.

[47] Y. Luo, L. K. John and L. Eeckhout, “Self-Monitored Adaptive Warm Up,” *International Journal of Parallel Programming*, To appear.

[48] Y. Luo, L. K. John and L. Eeckhout, "Self-Monitored Adaptive Warm-Up," *Proceedings of 16th Symposium on Computer Architecture and High Performance Computing*, pp. 10-17, October 2004.

[49] Y. Luo and L. K. John, "Efficiently Evaluating Speedup Using Sampled Processor Simulation," *Computer Architecture Letters*, vol. 4, pp. 22-25, 2005.

[50] Y. Luo and L. K. John, "Locality Based On-Line Trace Compression," *IEEE Transactions on Computers*, vol. 53, pp. 723-731, 2004.

[51] Y. Luo, J. Rubio, L. John, P. Seshadri and A. Mericas, "Benchmarking Internet Servers on Superscalar Machines," *IEEE Computer*, vol. 36, no. 2, pp. 34-40, February 2003.

[52] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, n o. 2, pp. 50–58, February 2002.

[53] A. M. G. Maynard, C. M. Donnelly and B. R. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 145-156, 1994.

[54] A. Milenkovic and M. Milenkovic, "Stream-Based Trace Compression," *Computer Architecture Letters*, vol. 2, pp. 14-17, 2003.

[55] A. T. Nguyen, P. Bose, K. Ekanadham, A. Nanda and M. Michael, "Accuracy and Speed-Up Of Parallel Trace-Driven Architectural Simulation," *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, pp. 39-44. April 1997.

[56] S. Nussbaum and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 15-24, September 2001.

[57] Open Source Development Labs. Database Test 2. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/ (accessed June 2005).

[58] M. Oskin, F. T. Chong and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 71-82, 2000.

[59] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation," *Proceedings of the 37th International Symposium on Microarchitecture*, pp. 81-92, 2004.

[60] D. Pelleg and A. Moore, “X-Means: Extending K-means with Efficient Estimation of the Number of Clusters,” *Proceedings of the 17th International Conference on Machine Learning*, pp. 727-734, 2000.

[61] E. Perelman, G. Hamerly and B. Calder, “Picking Statistically Valid and Early Simulation Points,” *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 244-255, 2003.

[62] PIN homepage, http://rogue.colorado.edu/Pin/ (accessed July 2005).

[63] A. R. Pleszkun, “Techniques for Compressing Program Address Traces,” *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 32-40, 1994.

[64] A. Poursepanj, “The PowerPC Performance Modeling Methodology,” *Communications of the ACM*, vol. 37, no. 6, pp. 47-55, June 1994.

[65] T. R. Puzak, “Analysis of Cache Replacement Algorithms,” Ph.D. dissertation, University of Massachusetts, 1985.

[66] A. D. Samples, “Mache: No-Loss Trace Compaction,” *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 89-97, 1989.

[67] C. D. Schieber and E. E. Johnson, “RATCHET: Real-Time Address Trace Compression Hardware for Extended Traces,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 21, pp. 22-32, 1994.

[68] T. Sherwood, E. Perelman and B. Calder, “Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications,” *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 3-14, 2000.

[69] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, “Automatically Characterizing Large Scale Program Behavior,” *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 45-57, 2002.

[70] SimpleScalar LLC. http://www.simplescalar.com/ (accessed July 2005).

[71] K. Skadron, P. S. Ahuja, M. Martonosi and D. W. Clark, “Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques,” *IEEE Transactions on Computers*, vol. 48, pp. 1260–1281, 1999.

[72] A. J. Smith, “Two Methods for the Efficient Analysis of Memory Address Trace Data,” *IEEE Transactions on Software Engineering*, vol. 3, pp. 94-101, 1977.

[73] R. Srinivasan, J. Cook and S. Cooper, "Fast, Accurate Microarchitecture Simulation Using Statistical Phase Detection," *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 147-156, March 2005.

[74] Standard Performance Evaluation Corporation, SPECweb99 Benchmark, http://www.spec.org/web99/ (accessed July 2005).

[75] Standard Performance Evaluation Corporation, SPEC CPU2000 Benchmark Suite, http://www.spec.org/cpu2000/ (accessed July 2005).

[76] R. Todi, "SPEClite: Using Representative Samples to Reduce SPEC CPU2000 Workload," *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, pp. 15-23, 2001.

[77] Transaction Processing Performance Council, TPC Benchmark C, http://www.tpc.org/tpcc/ (accessed July 2005).

[78] D. Vengroff and G. Gao, "Partial Sampling with Reverse State Reconstruction: A New Technique for Branch Predictor Performance Estimation," *Proceedings of the 4th International Symposium On High-Performance Computer Architecture (HPCA)*, pp. 342-351, 1998.

[79] T. A. Welch, "A Technique for High-Performance Data Compression," *Computer*, vol. 17, no. 6, pp. 8-19, June 1984.

[80] D. A. Wood, M. D. Hill and R. E. Kessler, "A Model for Estimating Trace Sample Miss Ratios," *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 79-89, June 1991.

[81] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 84-95, 2003.

[82] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337-343, 1987.

Vita

Yue Luo was born in Xiangfan, China, on December 11, 1972, as the son of Huanzhao Luo and Litong Yue. After completing his high school education at No. 1 Zhengzhou Railway Middle School, Zhengzhou, China, he entered the Department of Electronic Engineering in Tsinghua University, Beijing, China, in September 1990. He received the degree of Bachelor of Engineering from Tsinghua University in July 1995. He joined the graduate program for Electronics at Peking University, Beijing, China in September 1995 and obtained the degree of Master of Science in July 1998. He worked as a software engineer at ISD Co., Shenzhen, China until he entered the Ph.D. program in Computer Engineering at The University of Texas at Austin in September 2000. During the summer and fall of 2003, he interned at Sun Microsystems, Inc, working on a new web server benchmark. He is a student member of IEEE.

Permanent address: Henan Province, Zhengzhou City, Jingguang Zhong Lu
Tiedao Jiayuan 2 Haolou 49 Hao
China, 450052

This dissertation was typed by the author.