Processor design-space exploration through fast simulation

Taj Muhammad Khan

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

UNIVERSITE PARIS-SUD

ÉCOLE DOCTORALE : Informatique de Paris-Sud
Laboratoire : ALCHEMY, INRIA Saclay - Ile de France

DISCIPLINE : Informatique

THÈSE DE DOCTORAT

soutenue le 12/05/2011

par

Taj Muhammad Khan

Processor design-space exploration through fast simulation

Directeur de thèse : Olivier Temam
Co-directeur de thèse : Daniel Gracia-perez

Composition du jury :

Président du jury : Smail Niar

Rapporteurs : Babak Falsafi
Nigel Topham

Examinateurs : Frédéric Petrot

University of Paris Sud-11

Doctoral Thesis

Processor Design Space Exploration Through Fast Simulation

Author:

Taj Muhammad Khan

Supervisors:

Prof. Olivier Temam

Dr. Daniel Gracia-Pérez

May 13, 2011

Abstract

Simulation is a vital tool used by architects to develop new architectures. However, because of the complexity of modern architectures and the length of recent benchmarks, detailed simulation of programs can take extremely long times. This impedes the exploration of processor design space which the architects need to do to find the optimal configuration of processor parameters. Sampling is one technique which reduces the simulation time without adversely affecting the accuracy of the results. Yet, most sampling techniques either ignore the warm-up issue or require significant development effort on the part of the user.

In this thesis we tackle the problem of reconciling state-of-the-art warm-up techniques and the latest sampling mechanisms with the triple objective of keeping the user effort minimum, achieving good accuracy and being agnostic to software and hardware changes. We show that both the representative and statistical sampling techniques can be adapted to use warm-up mechanisms which can accommodate the underlying architecture's warm-up requirements on-the-fly. We present the experimental results which show an accuracy and speed comparable to latest research. Also, we leverage statistical calculations to provide an estimate of the robustness of the final results.

Contents

Chapter 1

Introduction

The ubiquity of microprocessors is witness to their indispensability in our daily lives. They can be found in almost everything; from processors humming in our computers (desktops, laptops, mobiles, etc.) to microcontrollers embedded in dishwashers, airplanes, ATMs, they are impossible to avoid.

This omnipresence has generated a reliance on their use and as a result fueled a growing number of expectations. Be it in terms of performance, power budget or economic viability, processor makers are under continuous pressure to innovate and improve on their offerings. This competitive nature of the market drives the design teams to deliver more performance in very strict deadlines. The fact that a microprocessor is a highly complex product often involving multiple teams of many persons in a multi-stage process usually spread over a long duration does not make things easier.

Figure 1.1 depicts a typical microprocessor design cycle in chronological order. Based on the market expectations or the clients' needs a chip maker gathers requirements for what functionality to be included in its next generation processors. This requirements gathering can be based on market surveys (its own or third parties') or on clients' feedback. A set of applications, called *benchmarks* are gathered to repre-

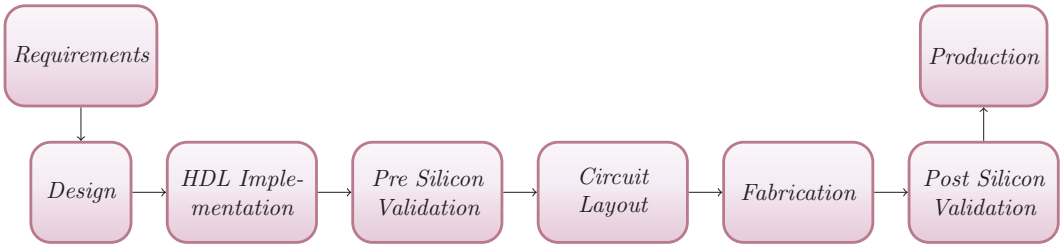


Figure 1.1: *Processor Development Life-cycle.*

sent this set of requirements and are used, from hereupon, to gauge the performance of new architectures.

These requirements are then fed to the design team which works out the changes to make to the existing chips to meet those specified needs. These changes may occur in the form of modification of the existing architecture components or addition of completely new modules. They may also happen by adding new instructions to the existing ISA. This is an iterative phase in which new features are tried and, if found unsuitable, modified many times. This design phase is the core phase of the process and it is here that important decisions, about which features to include, take place.

Once the initial design has been finalised, it is implemented in a low level HDL (Hardware Description Language), e.g., Verilog or VHDL, and tested against different functional and timing constraints.

It is then subjected to rigorous testing using a comprehensive series of test cases to verify that the design is always in a reliable known state starting from a viable input. All possible combinations of inputs are tested for a step by step execution and the state of the chip is verified after each step to ensure correctness.

Once the design is certified to pass these test cases, it is promoted to the next stage where the layout of the different components is optimised on the die. Some of the factors taken into consideration are: to minimize communication delay, to minimize power consumption.

After the detailed design has been laid out for the chip, it is sent to the fabrication plant where the actual die is taped out.

The actual chip prototype thus manufactured is again subjected to a thorough testing procedure to detect any defect during the manufacturing process or in case any bug may have escaped pre-silicon validation.

After the chip maker is sure that the resulting product is error free and meets all the design specifications, it is produced in large quantities and sent to prospective customers.

Manufacturing a processor is a long, involved and costly process. This is made evident by the fact that despite a large and burgeoning market, there are only a handful of players who can design and market a full processor and even fewer are those having the ability to fabricate them. The complexity and the resource requirements increase by orders of magnitude as the process advances in stages. A bug discovered at, or after, the fabrication stage can cost a company a lot in terms of revenue and reputation as Intel Corporation and Advanced Micro Devices, Inc., discovered both at their expense in 1995 [85] and in 2007 [40] respectively.

Thus simulation has emerged as a vital tool to ameliorate the design process by testing the performance of the design at various stages and detecting and correcting any error in the process. Depending on the abstraction level and the detail modeled, one can have access to different kinds of information in the simulation process. This is shown in Figure 1.2. A processor can be modeled analytically by taking a very abstract view of its complexity and thus have a bird's eye view of the design landscape without bothering with the details. Or it can be tested as a prototype made in the foundry simulating each detail exactly as it would be in the final product. Between these two extremes lie a range of simulation methodologies offering different capabilities that a user can choose from based on his/her needs. As depicted in Figure 1.2 the time required to simulate a program on a given processor configuration

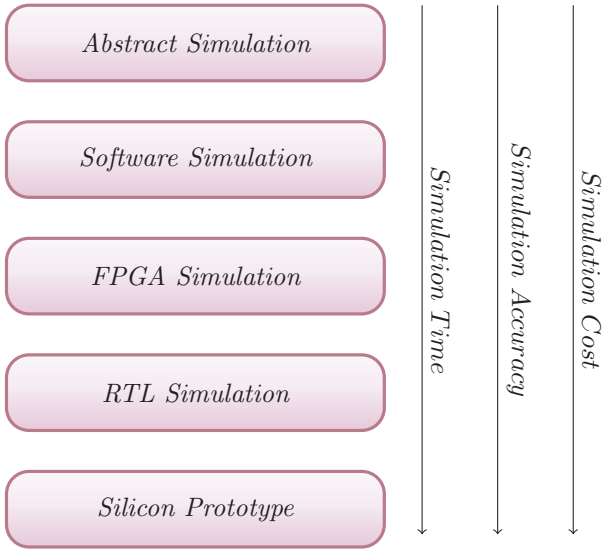


Figure 1.2: *Levels of Abstraction for Simulation.*

increases as we try to model more and more detail and so does the simulation cost which may be in terms of money or man-hours. Ideally, one would like to have a simulation methodology which delivers a maximum of accurate results at a minimum of cost and in a minimum of time. Thus the job of a simulation expert is to minimize the cost and time functions, and increase the accuracy. This is not always possible. Conventional wisdom in the industry says that from the three aspects of *cost, accuracy and time*, one can pick only two. And it is a fact that at each level of simulation, an architect makes a compromise on one or more of these parameters.

As the stakes are much higher in later stages of the process, it is highly desirable to detect and correct the design flaws in the early stages of the process. Furthermore, in the earlier stages of processor design cycle, major changes can be made easily having far reaching effects and benefits.

Figure 1.3 takes Figure 1.1 and expands the second step. We see that during the design phase, the specifications of the architecture are continuously being changed based on whether it meets the desired functionality or not. Software simulators

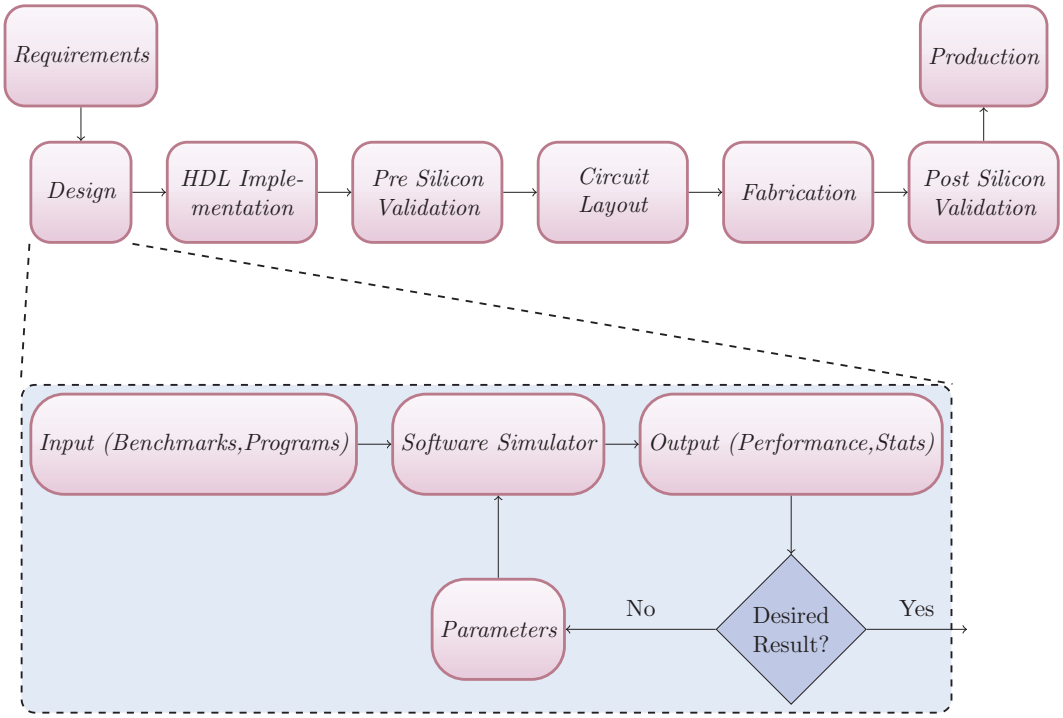


Figure 1.3: *Software Simulation in the Design Process.*

are the tools employed in this process to develop rapid functional prototypes of the processors and test their functionality. A processor simulator is fed the configuration of the architecture and the program to execute. It outputs the result in terms of the performance achievable for those parameters. These results are used as feedback to configure and test new architectures.

Using a software simulator, a processor can be modeled either at the *functional level*, which verifies the functionality without giving any detail about the timing information, or it can be modeled at *cycle level*, which models the timing and microarchitecture detail, to gain insights into the performance in terms of execution time. While the functional simulation is faster in terms of execution, it does not give any information about the performance metrics. We need the cycle level, also termed as *detailed*, simulation for that. Note: In this document we use the terms *simulation,*

cycle level simulation, detailed simulation, and *performance simulation* indifferently unless stated otherwise. *Functional simulation* will always be mentioned explicitly either by its own name of by referring to it as *fastforwarding*.

Detailed software simulation is preferable to other methods of simulation as it can model more detail and hence be more accurate than analytical modeling at relatively modest increase in cost and is much cheaper in terms of development than the FPGA simulation with relatively little loss in detail. This flexibility in development and simulation of detail comes at a price however. Software based detailed execution driven simulation can be many orders of magnitude slower than executing the same program on a real machine. KleinOsowski *et al.* [55] measured that SimpleScalar [12] executes 3000 cycles of the host machine to simulate 1 cycle of the simulated architecture. This made it 3 orders of magnitude slower. At this rate they estimated that the benchmark `188.ammp` would take 16 months to do a detailed run of its *ref* input set.

To explore all the microarchitectural design space, the architect has to test different combinations of values for all of the parameters. The number of parameters to explore and the set of values for each of them makes the design space difficult to explore in its entirety. For each combination of parameters, the user has to run many programs comprising the benchmark suite. Exploring all the combinations of all the parameters for a processor may need a program to be executed millions of times. Despite these inordinate simulation times and the immensity of the design space, the need for simulation makes it unavoidable and thus has forced researchers to seek methods to circumvent these seemingly insurmountable challenges. Attempts have been made to reduce simulation times as well as to prune the design space to a reasonable size.

Techniques to cope with intractable simulation times include data set reduction [55], intelligently skipping the design space [17, 42] and sampling [92, 110]. All these

techniques are orthogonal and are usually used together to test a maximum number of architectures in a reasonable time.

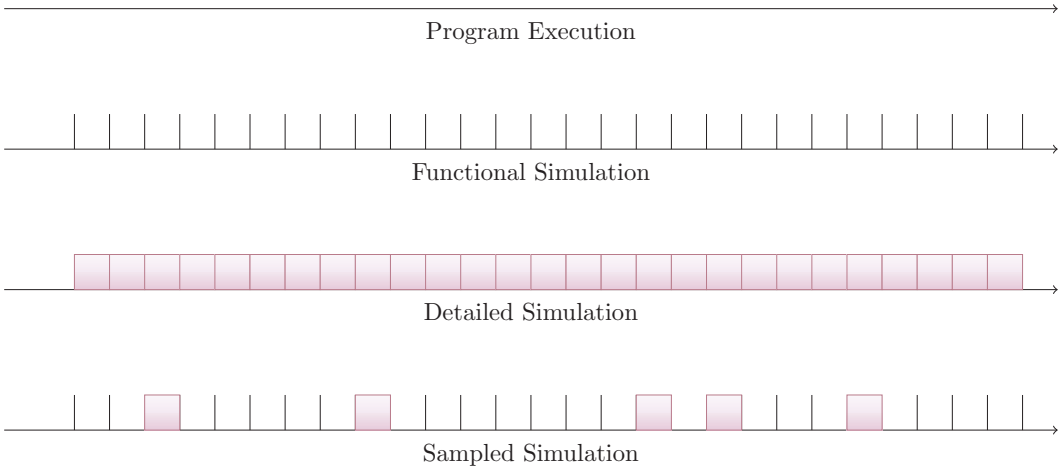


Figure 1.4: *Software Simulation Techniques.*

Sampling accelerates the design space exploration by reducing the execution times of individual program simulations. Instead of measuring the performance of the whole program in detailed simulation mode, Sampling, as the name suggests, measures the performance only of selected portions of interest. It then tries to reconstruct the performance of the whole program via the performance of these "samples".

Figure 1.4 depicts the idea of Sampling. The execution of the program is divided into intervals of equal number of executed instructions. Both *functional* and *detailed* simulators will execute these intervals in their respective fashions. However, for *Sampling*, a simulator should be able to switch between the two. As shown in the figure, the simulator would *fast forward* most of the program in functional mode and simulate in detail only intervals which it thinks may help it calculate the performance of the program.

By executing most of the program in the *fast* functional mode, Sampling reduces the simulation times considerably. For this reason, as well as its accuracy, it has

become one of the most popular tools in processor design space exploration. In addition to being fast and accurate, an ideal simulation methodology shall also be transparent to the user and compatible with frequent software changes. While the first two desired properties are the focus of most studies, we shall hereby provide the motivation for the importance of the latter pair when considering the state-of-the-art sampling techniques. For an end-user, who during design exploration would simulate many configurations, the simplicity of use is an important issue.

SimPoint [92] is a phase-based, representative, sampling technique. After doing a phase analysis of the program to choose the representative samples, it then simulates them in a second go. It does not require any changes to be made to the simulator as such making it an easy-to-use technique. However, SimPoint does not take into account the effect of cold simulator state on Sampling. This leaves the user with the task of choosing the warm-up mechanism whose effect on the simulator speed and accuracy can be non-trivial. Secondly, samples simulated by SimPoint depend on the prior choice in the phase analysis part. Any modification of the program code (or sometimes even the input) would result in a different phase behavior and will require the phase analysis to be done again. This way, the SimPoint mechanism is not transparent to software changes. In an environment with frequently modifying software (embedded hardware/software co-design), a simulation strategy transparent to such modifications is highly desirable. Pereira *et al.* [81] try to address this issue by merging the phase analysis part with the simulation but their warm-up mechanism is effected by changes in hardware and software.

SMARTS [110] is another popular sampling mechanism. SMARTS chooses its samples randomly and thus does not require prior phase analysis. This makes it transparent to software changes, i.e., modifications to software code do not affect sample choice. Secondly, SMARTS proposes a warm-up mechanism compatible with its sampling strategy. It warms the state of the simulator during the functional

mode. This renders the warm-up compatible changes in hardware and software as opposed to on-line SimPoint [81]. However implementing the functional warm-up means modifying the functional simulator, i.e., all state-storing structures in the detailed simulator have to be copied in the functional mode and updated. This can require considerable effort in terms of development from the programmer. This also exposes the user of the simulator, who might be different from its developer, to the internal details of the simulator. Sometimes it might not even be possible to warm some structures in functional mode, i.e., there are structures which record time-based information [41]. A functional simulator, having no notion of elapsed time or cycles, cannot know the information needed to update such structures. The same issues are inherited by Kihm *et al.* [54] who try to exploit program phase behavior for sample selection but keep up the functional warm-up.

Keeping in view the above discussion, there is a need for a sampling technique that is transparent to code changes (like SMARTS[110]) and at the same time easier to implement and use, i.e., requires a minimum of simulator modifications (as in SimPoint[**?**]). At the same time it should adapt a warm-up method that is compatible with frequent software/hardware changes.

To have a more practical warm-up strategy, we propose using the detailed simulation as warm-up. This has the advantage that it requires no simulator modifications unlike functional warm-up [110] and also that the time-based structures are updated automatically as detailed simulation mode keeps track of time. To make the warm-up methodology compatible with hardware changes, we are inspired by the recently proposed adaptive warm-up methods [69]. By keeping track of how much warm-up we need each time we are to take a sample, we not only avoid unnecessary detailed simulation (which is time consuming) but also achieve a good accuracy.

We apply this adaptive warm-up with both the Statistical as well as Representative Sampling. The former, as discussed about SMARTS[110], has the advantage of

being unaffected by the software modifications. To render Representative Sampling compatible with frequent software changes, we adapt the online phase classification presented by Pereira *et al.* [81]. While it may appear, in the latter case, that it is a simple combination of on-line SimPoint [81] and SMA [69], we shall show in the following chapters that SMA is not compatible with SimPoint as it renders the start of sample variable which is undesirable in SimPoint.

In both modes, we emphasize transparency to software and hardware changes as it is an important issue when someone actually wants to use the technique. An ideal approach should adapt itself to the characteristics of the program as well as the architecture to report back accurate performance results with a minimum intervention from the user. The details of our work, the experimental methodologies and results and their analyses are described in the document that follows. It is organised as follows:

Chapter 2 gives an overview of the field of *Computer Architecture Simulation*. We expound the idea of Simulation as an important tool in the process of design and testing new and better architectures. We detail its benefits and shortcomings. We start with the different methods and platforms employed in the industry and academia to simulate. A brief discussion of each is provided. We also describe a list of simulators being used. The discussion is narrowed down to software simulation and Sampling. Attempts to use Sampling to estimate the performance of architectures in the past are detailed with their pros and cons. Other aspects such as warm-up and usability are also taken under consideration.

Chapter 3 presents Transparent Representative (Stratified) Sampling. After a brief introduction we present a discussion of related topics such as mechanisms for *Program Phase Classification*, *Phase Prediction* and *Warm-up*. We then put all the pieces together and present an overview of our Sampling mechanism. We, next, present the platform used for experimental verification and the benchmark programs

used. Then follows a presentation and discussion of our results in term of *Performance Estimation Error* and *Percentage of Detailed Simulation*, i.e., Simulation Time.

Chapter 4 presents Transparent Random Sampling. It is organised in the same way as the previous ones, i.e., starting with an introduction, followed by discussion of the statistical concepts used. Later we present a discussion of *sample selection* and *Warm-up*. We also delineate our mechanism for keeping the amount of detailed simulation in check. After presenting our experimental platform, we detail the results and discuss their merits.

Chapter 5 concludes this document by recapitulating the important findings and insights gained during this work. It continues the discussion by proffering future directions that can be pursued following this work.

Chapter 2

Related Work

Moore's law [72] states that the number of transistors on a processor shall double every 24 months. Based on this assertion, , for the past few decades, the leading processor manufacturers have been churning out performance-enhanced versions of their processors every year. These advances in performance stem from a research ecosystem of collaboration between the industry and the academic groups working on improving the design and performance of current and future systems. As stated in Chapter 1, the industry uses simulation as an important tool to rapidly develop and verify designs during different steps of the processor manufacturing process. Due to lack of resources, the academic community relies even more heavily on the simulation process to test new architectural concepts. The panelists of a workshop in 2001 [95] observed that the number of simulation based papers in ISCA (International Symposium on Computer Architecture), a premier computer architecture conference, increased from 7.1% (2 out of 28) in 1973 to 87% (27 out of 34) in 2004. They questioned the justifications for this increased reliance on simulation and whether it was happening at the expense of other techniques. Nevertheless, this gives an idea of the importance of simulation in the eyes of the researchers.

A simulator tries to mimic and reproduce, as closely as possible, the behaviour

of the machine it is imitating. Simulation has become the de facto preferred method of evaluating microarchitectures because usually the researchers: do not have access to real hardware, do not have enough money to manufacture their own products, or do not have the time or manpower to invest in a long and complicated process.

When it comes to simulation, several questions arise. How accurate is the simulator? How faithful are the performance measures reported to the actual machine performance? Though it's difficult to answer this for future architectures, there have been attempts to model existing processors and verify a simulator's correctness. Black and Shen [10] developed a trace-driven model for the IBM/Motorola PowerPC 604 processor and compared the performance of the simulator against that of the actual chip, measured using hardware-embedded counters. Using a series of generated test cases and a rather strict criterion of success (exact CPI match), they found that for the test suite with best results only 51% of the test cases passed initially. They qualified three sources of errors. *Modeling errors* resulting from a buggy implementation. *Specification errors* occurred when the developer was provided with or assumed wrong specifications. *Abstraction errors* were caused by the developer implementing a feature at a higher level of abstraction. By systematically eliminating these sources of errors and slightly relaxing the validation criterion, they were able to increase the percentage of passed test cases to 84% on average. The best test suite passed all of its test cases. Using some real-world benchmarks, they reported an error of 4% in performance estimation. A similar study was conducted by Desikan *et al.* [21] in 2001. They modified the SimpleScalar simulator to model an Alpha 21264 processor. Using some synthetic benchmarks, they found an average discrepancy of 74% between the performance reported by the simulator and that of the real chip. A bit of reverse engineering and modeling a lot more detail helped them reduce the average error to 2%. However for real world SPEC CPU2000 benchmarks they still reported an error of 18%. Using the generic *sim-outorder* model,

configured as closely to the Alpha 21264 parameters as possible, resulted in an error of 36% for the same benchmarks. So modeling more detail helps get a more precise picture of processor performance.

The more detailed our model, the more faithful our simulator is to the original processor. The catch is that the more detail we model, the longer get our simulation times. It is desirable, especially in the early stages of the design process, to have a fast evaluation methodology in order to explore a maximum portion of the design space. This said design space consists of all architecture models that can result by varying different values for the architectural parameters taken into consideration. One popular simple simulator [12] has 40 such parameters. Assuming 2 values for each parameter will result in a space consisting of 2^{40} configurations to evaluate. Clearly, it is impossible to exhaustively search this space. Therefore, the need for a fast mechanism of exploration cannot be overstated. This brings to spotlight the basic dilemma facing every computer architect: What is the ideal compromise between the accuracy of the model and the simulation time?

In the December 2001 workshop [95] organized by the US National Science Foundation, renowned names in Computer Architecture Simulation were brought together to brainstorm on major issues in the domain. They identified problems in key areas and made suggestions to improve the situation. **Absolute vs relative accuracy.** The panelists agreed that, when doing design space explorations, absolute error in performance measurement is rather less important than the relative trends in performance as we vary the parameters. As long as a simulator catches these relative changes in performance correctly, we can sacrifice the absolute accuracy for speed. **Simulators vs simulation frameworks.** They unanimously favored simulation frameworks over monolithic simulators. Their modularity and portability properties make simulation frameworks attractive to developers. **Benchmarks.** They stressed the need to better characterize existing benchmarks to show how much of

the application behavior space they actually represented. They propounded the use of micro and synthetic benchmarks for their ability to better isolate individual program behaviors and to be parametrized respectively. **Abstractions.** They advocated increased use of more abstract analytical models to help fast exploration of design space. **Evaluation metrics and validation.** They emphasized the need for new and more descriptive metrics to capture program behavior and accessible statistical tools to verify their robustness. They also lamented a lack of infrastructure and encouragement of the replicability of the published experiments.

In a similar gathering in ISPASS (International Symposium on Performance Analysis of Systems and Software) 2004, a panel of experts from the architecture simulation community reunited [112] to assess the state of the art and found the field lagging behind in several important areas. They found an increased reliance on execution driven cycle-level simulation at the expense of analytical and statistical techniques. They made five "key recommendations":

- Need to put more thrust in multiprocessor and full system simulation.
- Need to propose more efficient simulation methods.
- Need to explore fast alternatives to cycle-accurate simulation.
- Need to increase representativeness and decrease redundancy in benchmarks.
- Need to add robustness and statistical validation to simulation methodology.

Creating a detailed and validated simulator can be an arduous process. Modeling the exact detail of a chip is avoided for the following reasons: it is extremely resource consuming in terms of man-months, most manufacturers do not publish the detailed specifications of their chips for corporate strategic reasons, once modeled the simulator is difficult to modify if a user wants to adapt it to his/her purpose, and, a complicated design is difficult to validate as well. In [5] Todd Austin tells his

experience of developing an ARM [3] target for SimpleScalar [12] where validating the simulator took *more* time than developing it.

Since it is time consuming and complicated to exactly model a processor design in detail, many simulators make abstracting assumptions about different components as long as they do not violate the results too much. This has the advantage of simplicity and reduces development time. Secondly, the simplified model is easy to understand and modify facilitating adoption by the community. Another plus point is that while a specific architecture may become out of fashion with time, a generic model continues to be used and can be adopted to one's needs. This can be seen by the adoption of popular simulators [12, 70] in the research circles. SimpleScalar [12] tools claim, on their website, to be used for "more than one third of all papers published in top computer architecture conferences" in the year 2002.

The problem of architecture simulation is a multi-faceted one. The design space is expansive. Benchmarks multiply and increase in size as consumers demand a rich and intensive computing experience. The complexity of processors is increasing making old simulators obsolete and newer ones difficult to model and slower in execution. Lack of standard criteria and platforms makes it difficult to compare results. There is a concerted effort on the part of the researchers to tackle this problem from all its angles. Efforts are being made to reduce the design space to a manageable size and traverse the space intelligently. Benchmarks are being analyzed for redundancies and their size curtailed with minimal effect on their characteristics. Techniques are being developed to make the simulators and simulations of individual programs go faster. After presenting an overview of benchmarks and tools used, we discuss all these efforts in the following sections.

2.1 Benchmarks

The choice of benchmark programs affects the run times and result interpretations for simulations. Simulating wrong or non-representative benchmarks can cause an architect to make design choices which will perform poorly in real world. On the other hand, executing large benchmarks with overlapping characteristics can cause redundant simulations and waste of costly time. This section presents a preview of common programs used in the research community.

Standard Performance Evaluation Corporation [97] first released its set of SPEC CPU benchmarks in 1989. Since then, after going through five generations (1989, 1992, 1995, 2000, 2006), it has been one of the most commonly used benchmark suite in the architecture community for measuring single-threaded desktop performance. The size and diversity of the benchmarks have increased steadily reflecting the advances in CPU performance and the needs of the users. SPEC CPU89 had 10 benchmarks with an average length of 2.5 billion instructions; SPEC CPU2006 has 29 programs with an average size of 2.2 trillion instructions each. The SPEC CPU benchmarks are grouped in two main groups: the *int* and *fp*, and are representative of different fields, from business and scientific applications to quantum chromodynamics, weather prediction and linear programming.

Phansalkar *et al.* [82] do an analysis of SPEC CPU benchmarks over four generations, from 1989 to 2000. They characterize the benchmarks in terms of their instruction count, branch characteristics, data locality and instruction-level parallelism (ILP). They observe that though SPEC CPU has kept some of the programs same over different generations, the behaviour of these programs sometimes changes with time. In the SPEC CPU2000 suite they cite the examples of `swim` and `applu` having more conditional branches in loops and worse temporal locality, respectively, from their ancestors. They use principal component analysis (PCA) to reduce interactions between the parameters and then use k-means clustering to cluster similar

applications. Redundancy in benchmark suites, different programs having same characteristics, is undesirable as simulating the same kind of applications does not furnish any additional information. They observe that over the four generations, the redundancy in SPEC CPU benchmarks has also increased. This is confirmed by [83] who found the SPEC CPU2006 suite to be 50% redundant, i.e., one can simulate only half of the suite and have almost the same information as would be obtained by simulating the whole.

SPEC workloads are geared towards desktop and server systems. MiBench [38] were proposed by the University of Michigan as an alternative set of benchmarks targeting embedded systems. The applications are divided into six groups targeting six segments of the embedded market, namely, automotive control, consumer devices, office automation, networking, security, and telecommunications. Guthaus *et al.* [38] analyzed the characteristics of these programs and compared them to the SPEC CPU2000 ones. They noticed that these benchmarks showed much more variability than the SPEC CPU2000. This is in consistence with the difference between the nature of the two markets, the embedded one featuring many ISAs and architectures. While the branches were rather predictable for both benchmark suites, they found that MiBench programs have lower memory requirements and the percentage of cache misses drops low for cache sizes 2-4 times smaller than those of SPEC CPU2000.

2.2 Simulators (Stand-alone and Full System) and Simulation Infrastructures

The simulator finds itself at the heart of the simulation process. In the early days of architecture simulations everyone would create a simulator to publish results. But as the complexity of processors increased (out-of-order, branch prediction, etc.),

so did the effort to build simulators. Furthermore, the prevalence of simulated performances created a need to compare performances. These reasons together forced the adoption of standardized and reusable platforms for publishing results. This resulted in simulators being developed and validated by communities for the purpose of easing the simulation process.

Simulators can either act on traces of programs or they can take a binary and execute each instruction one by one. *Trace driven simulation* uses a time ordered sequence of events generated by a prior detailed simulation. It has the advantage of speed and the fact that a trace, once generated, can be used as many times as needed. The drawback is that it lacks certain details. It cannot model wrong path executions. Also, a modification of program necessitates a regeneration of the trace. It is quite easy to instrument [78] binaries for trace generation. *Execution driven simulation*, on the other hand, executes the instructions of a program as they would be executed by a real processor. It models pipeline flushes in case of branch mispredictions and cache pollution due to wrong path executions.

SimpleScalar tool set [12] is the most widely used simulator for simulating single-threaded performance. SimpleScalar consists of multiple simulators and the tools which help simulate programs on them. These tools include a compiler, an assembler and a linker. It models the Alpha and PISA (Portable Instruction Set Architecture) instruction sets. The simulators consist of sim-outorder (a detailed out-of-order simulator implementing a five stage pipeline, branch prediction and memory hierarchy), sim-cache, sim-cheetah (simulators for caches) and sim-fast (a functional mode fast simulator). There exist numerous extensions for SimpleScalar and, due to its simple, generic architecture, it is widely used.

SESC [75] is another more recent event driven simulator which models a MIPS instruction set architecture. It divides the execution into an emulator and a timing simulator. The emulator executes the instructions functionally and then sends an

object with relevant information to the timing simulator which then calculates how much time this instruction would take through the pipeline. It uses MINT [106] to emulate system calls. It can simulate multiprocessor systems with interconnection networks.

Stand-alone simulators usually do not model the full memory/disk hierarchy and the I/O subsystems. Full system simulators model the complexities of real world environments and help model the operating system effects. Cain *et al.* [13] showed that operating system effects can make a difference of 100% for the SPEC CPU 2000 benchmarks. Simics [70] is a full system simulation platform than can simulate UltraSPARC, x86, PowerPC, MIPS architectures. It can boot Linux and Windows. Simics was developed by VirtuTech which got bought by Wind River Systems which now belongs to Intel Corporation.

COTSon [2] is another full system simulation infrastructure developed at HP Labs. It also decouples the functional and timing simulations. For functional simulation it uses AMD's SimNow simulator. It can model multiprocessor systems with a full unmodified operating system running on it.

The complexity of modern processors has made it difficult to model state of the art systems while starting from scratch. The same complexity, as it translates into coding complexity, also makes it difficult to modify existing simulators. This has forced researchers to think in terms of *modularity* and *code reuse*. Modular simulation environments have emerged where the user can develop only the module that is of interest and reuse the other existing parts. ASIM [32] is such a framework which contains modules with well defined interfaces for inter-module communication. It also provides a set of tools to manage software components. Developers can develop and test either stand-alone modules or plug them into the system to get an idea of full system performance effects. The Liberty Simulation Environment (LSE) [101] is another such environment. It lets developers map hardware components as

software modules and define connections between them. Using already developed modules significantly reduces development effort on the part of the user. UniSim [4] builds on LSE proposing distributed control signals, TLM simulation and a library of reusable components. Reduction of development effort because of module reuse also encourages replication of results and fair comparisons. This can give surprising results as demonstrated by Gracia Pérez *et al.* [37].

Then there are tools that are not exactly full simulators but help simulation. Pin is a dynamic binary instrumentation tool which allows inserting user defined analysis code in arbitrary locations in a program binary. This way, a user can insert functionality to collect profile data during the runtime of a program. Wattch [11] models power consumption in a processor core by modeling the thermal properties of array structures, fully associative content-addressable memories, wires and combinational logic, and clock buffers. Similarly, Cacti [73] can model timing, area and power aspects of modern caches.

2.3 Design Space Exploration

As stated in the introduction to this chapter, computer architecture simulation consists of finding out the right values of the different parameters for all the modules comprising the processor such that the resulting machine would execute all target applications in optimal time. Easier said than done, the combinations of parameter values and programs to explore skyrocket quickly. On top of this, the slow detailed simulation speeds make the task even more daunting. Due to the immensity of the problem, it is impossible to be tackled in its entirety. Therefore, researchers attack it piecewise. Usually, this approach divides the problem along three broad axes, namely, *traversing the design space intelligently*, *reducing the number and size of benchmarks*, and *accelerating individual simulation executions*. This section gives a broad view of all these aspects, especially focusing on the last one as it forms the

core of this thesis.

2.3.1 Managing the Design Space

A processor is composed of many modules. Each of them itself is a complex product of many carefully tuned parameters. There is the pipeline length, buffer sizes, issue and retire widths; memory hierarchy block sizes, associativities, latencies; bus widths and wire delays; a horde of predictors and prefetchers with histories and table sizes. The best configuration could be for any set of values of all these parameters. To check them all would mean to simulate and analyze all the programs for every combination of these parameters. Doing detailed simulations for such a design space could take many lifetimes. The first realization which helps reduce the size of the problem is that not all the parameters values have effects noticeable enough to be considered important.

Sensitivity analysis. To avoid exploring design space naively, we must identify parameters that affect most the targeted performance metric. *Sensitivity analysis* consists of modifying the values of different parameters one at a time and seeing how the change affects the final output. It can give insights about the importance of parameters and that of their interactions. Skadron *et al.* [94] study the effect of branch prediction, cache size and the instruction window size on the *instructions per cycle (IPC)* of SPEC CPU95 programs on a modified version of SimpleScalar. They found out that increasing the RUU (Register Update Unit) size beyond a certain point has no effect on the performance because the number of instructions active at a given time is limited by the branch predictor accuracy. They also observed that the performance is more sensitive to instruction cache size than to data cache size. If the size of the instruction cache is small, increasing the data cache size has no effect.

Statistical techniques. Statistical methods have proven to provide acceptable solutions to untractable problems in many fields. Computer architects have not hesitated to resort to their help whenever they could. Yi and Lilja [116] use their knowledge of the working of the processor to form intelligent heuristics about which parameters to remove from the sensitivity analysis. Once they have the number of parameters reasonably reduced, they proceed to see the effect of each parameter on the final output. They used ANOVA (analysis of variance) [67] to determine the effect of each parameter on the final output. To identify the sensitivity of a metric to N parameters and their interactions, the ANOVA technique requires 2^N simulations. In this way they identify the most important parameters. Once the search space is small it can even be explored exhaustively. In [117] Yi *et al.* use the Plackett and Burman statistical method to identify parameters which contribute most to the final output of the performance. Plackett and Burman requires 2N simulations for N parameters but fails to quantify the effect of different parameter interactions on the performance. They identify the most important parameters by Plackett and Burman, and then use the ANOVA technique to quantify the effect of their interactions.

Oskin *et al.* [76] use HLS, a statistical modeling framework, to explore the design space. They gather statistical profiles of application binaries by running them on a modified SimpleScalar. These profiles consist of data about basic block size and distribution, dynamic dependence distance between instructions, cache behaviour and branch prediction accuracy. Using this information, the framework produces synthetic instruction streams whose profiles match those of the original applications. These synthetic streams are much shorter in length and thus faster to execute. To explore the design space, they vary the design parameters in both software (basic block size, dynamic instruction distance) and hardware (cache miss rate, branch predictor accuracy, latencies). They found that for parts of the design space with high

branch predictor accuracy and high cache hit rates, the results for these synthetic streams matched those of the full applications. On the other hand, the performance degraded as these values got lower. They attribute this to the over simplicity of modeling cache hit rates and branch predictor hit rates as simple normal distributions. Using averages and standard deviations ignores the dynamic behavior of these parameters and their interactions that occur during the course of execution. Nevertheless, they recommend using the framework for a fast exploration of the ranges of design space where it performs well.

Eeckhout *et al.* [29] make a case for using statistical simulation in early stages of design space exploration. They profile the programs in terms of their *instruction mix*, *inter-instruction dependencies*, *cache*, and *branch predictor behaviour*, and generate a synthetic instruction stream matching the original profile. They demonstrate that for *uniprocessor performance modeling*, *power modeling*, and *system evaluation*, statistical simulation does a good job of predicting the relative performance changes when varying architectural parameters.

Joshi *et al.* analyze different statistical simulation techniques to verify their effectiveness in determining absolute and relative performances. They found that statistical simulations can be useful to identify processor performance bottlenecks. They also remark that while it takes a very detailed model to achieve useful absolute accuracy, statistical simulation techniques show good relative accuracy and can be used to track design changes. They stress the need to model data and control dependencies to ameliorate the statistical models.

Analytical modeling. In [50] Jouppi developed a model for determining instruction-level and machine parallelism for different programs. He partitions the interaction between the machine parallelism (MP) and benchmark parallelism (BP) into two areas:

if(BP $>$ MP), performance is limited by the machine parallelism,

if(MP > BP), performance is limited by the benchmark parallelism.
He developed a first order model to determine this parallelism but finds that using *averages* for different parameters makes for poor performance. Instead he recommends taking into account the non-uniform variations in benchmark and machine parallelisms over the course of time.

In 2002 Noonberg and Shen [74] described a theoretical model for Superscalar processor performance. They build on Jouppi's idea of parallelism in program and machine by using probability matrices. The probabilities are intended to capture the variability in the program and machine parallelisms. The probabilities for the program parallelism matrix are functions of data and control dependencies while those of the machine parallelism take into account parallelism due to branch prediction accuracy, the fetch and decode mechanism, and the processor issue width. They model the IBM RS/6000 processor and modify the number of its functional units. Running selected SPEC CPU92 *fp* and *int* benchmarks, their model estimates the IPC within -0.6% and +22.0% of the real IPC.

Karkhanis and Smith [51] proposed an analytical model for a superscalar processor core. They develop a model for pipeline performance under ideal conditions i.e., perfect branch prediction, no cache misses, etc. This ideal performance is then augmented with equations developed for penalties incurred by the branch misprediction and cache miss events. They use trace driven simulation to calculate the distribution of occurrence for these *miss-events*. Using their analytical model for the processor, they were able to predict the performance of the SPEC *int* benchmarks with an average error of 5.8%.

Predictive modeling. Machine learning techniques are also called in to help traverse the design space. This is done by running detailed simulations for multiple configurations. These configuration parameters and their results are then used to train a predictor. This predictor is then used to predict the performance for new

configurations.

In [43], Ipek *et al.* use artificial neural networks (ANNs) to explore the architectural design space. They devise an artificial neural network with a 16 unit hidden layer to train on SPEC CPU2000 input programs and with only 1% simulation of the design space claim to predict the rest of the design space *for the same applications* with 98-99% of accuracy. They use SimPoint to reduce the time of the input simulations and claim that the noise-resistant nature of the neural networks lets them perform good despite error introduced by SimPoint.

In [18] Cook and Skadron use GPRS (Genetically Programmed Response Surfaces) to predict the design space performances. They use genetic programming principles to create and train non-linear polynomial approximation functions from collected architectural performance data. These functions (GPRS) are then used to predict the performances for new configurations. Using 1% of the design space for training, they can predict IPC for other configurations with 2% mean percentage error.

Khan *et al.* [52] take this a step further and use the ANNs trained for a set of programs on a subset of configurations. These ANN models are combined with simulations for a new application on the same configurations. The idea is that the neural model will classify this new program with one of the previously seen programs and use the training for that previous program combined with the new data to predict the performance of the new application. Their neural network consists of a hidden layer of 10 neurons. They use the SESC simulator with SPLASH-2 and SPEC CPU2000 benchmarks executed in TLS (Thread Level Speculation) mode. In predicting the energy-delay metric for a new application they report a prediction error ranging from 3.1% to 4.9%.

Dubach *et al.* [24] propose an architecture-centric approach to train neural networks to predict performances over the design space. They use N program-specific

predictors composed of multi-layer perceptrons with a hidden layer of 10 neurons. They start with an off-line training phase where they recommend using 512 detailed simulations to train each program predictor. In the second phase they use 32 simulations of the new program to train its predictor and then combine the output of this new predictor with that of the previous predictors in a linear (regression) manner. This new function is then used to predict the performance of the new program in the design space. They report an average error of 7% and a correlation coefficient of 0.95. From a list of 3000 configuration they claim to find the best configuration with only 3 more detailed simulations.

In [14] Cavazos *et al.* use the ANNs to predict the effect of program transformations on speed-ups. They use a multi-layer neural network with a 5 neuron hidden layer. They use the UTDSP benchmarks to train the network. Randomly chosen transformations are applied to each program to create 64 versions of it. These are then used as training inputs for the neural network. Once trained, the model is presented with a new program and its performance for 4 different transformations. Using this limited information of 4 performances, they predict the new program's speed-up for the rest of the transformations with an error of 7.3% on average.

2.3.2 Workload Characterization

When doing design space exploration, it is extremely important to have a representative workload. Non-representative workloads result in loss of precious simulation time. Normally, benchmarks are supposed to provide a maximum coverage of the application space with a minimum of programs but that is not always the case. When discussing workloads there are two things to consider: 1) Which benchmarks to select and simulate, and 2) What set of inputs to use when simulating? There have been attempts to identify the most representative program-input pairs and to simulate them instead of all the combinations.

Eeckhout *et al.* [30] propose a method to analyze program-input pairs' characteristics and then select only the most representative ones for simulation. They analyze some 20 program parameters consisting of instruction mix, cache misses, branch prediction accuracy, etc, and 79 program-input pairs. They use the PCA (Principal Component Analysis) to transform the 20 program characteristics to 2-4 *components* and use these components instead. They then use *hierarchical clustering* to group the program-input pairs into clusters based on their similarities with respect to these components. They advocate simulating only the representatives from each cluster.

KleinOsowski *et al.* [55] try to reduce the simulation times of the SPEC CPU2000 benchmarks by reducing the input dataset sizes. They reduce the datasets by: 1) manipulating command-line parameters, 2) truncating input files, 3) creating new input files. They compare the similarity between the new and old data sets by comparing: 1) functional profiles of both runs, 2) the instruction mix, 3) cache miss profiles. They found out that while the functional profile of the SPEC CPU2000 programs run with reduced data sets sometimes differed from that with *ref* inputs, the instruction mix profile usually matched.

Phansalkar *et al.* [83] use PCA and hierarchical cluster analysis to identify the similarities between the SPEC CPU2006 benchmark programs on four different ISAs. They found the behaviour of the programs to be sensitive to the input sets. However, they also found out that 14 out of 29 programs were sufficient to capture most of the information. Seeing the lengths of SPEC CPU2006 benchmarks, it would be a considerable waste of time to simulate all this redundant code.

Yi *et al.* [114] found PCA and Placket and Burman to be better methods for benchmark subsetting than other methods. For these two methods, most of the programs' shortened versions estimated the CPI (cycles per instruction) and EDP (energy delay product) with an error of less than 5%.

Citron [15] deplores the use of partial SPEC CPU2000 benchmark suite even in reputed conferences like ISCA and MICRO. More importantly he points out the lack of adequate explanations for the missing benchmarks. He argues that some benchmarks are preferred over the others because of their portability and ease of compilation. In making some negative assumptions (slowdowns) about the missing benchmarks, he observes that the overall speed-ups reported by the published papers for partial subsets can be considerably reduced if the missing programs were taken into account.

In [113] Yi and Lilja make a survey of the field of computer architecture simulation. They discuss the most popular simulators and benchmarks being used by the scientists and the methodologies to set up and analyze simulation experiments. They also comment on the common simulation acceleration techniques stating the merits and demerits of each. At the end of their comprehensive document they complain about the lack of documentation on simulation methodologies in published papers which impedes reproduction of results, and the lack of care and explanation for the choice of simulator parameter values since they affect simulation results. They recommend more effort in improving the accuracy of the simulators and reduction of simulation times.

2.3.3 Simulation Acceleration

In this section we detail the third main axis of faster design space exploration which consists of reducing individual simulation times. Simulating a program on a simulator can be considered as the *inner most loop* of the design space exploration process. Any reduction in simulation times will have a direct impact on the processor design space exploration times. As a result, there have been a number of tentatives to reduce the execution times of program simulation be it by development of faster simulators or by crafting mechanisms to make the programs go faster. Below is an

account of different techniques that have been developed to make the simulations go faster.

2.3.3.1 Direct Execution

The fact that code is executed much faster on a real processor than on a simulator and that a simulator need not execute all of a program code makes the case for *Direct Execution*. Direct execution [57, 34] advocates the execution of the simulated program's code on the host machine in stead of the simulator. Chen [34] uses it to run the fast forward portion of their program on the host machine. However, they keep a record of the instructions executed and the branch predictions to later warm up the simulator's structures before they do detailed execution. Krishnan and Torrellas [57] also use direct execution to simulate superscalar processors. They use a simulator front end and a MINT [106] instrumented binary to execute instructions on the host processor. The instrumented binary generates information (opcode and register usage) which is logged in a 512 entry *Interface Window*. These events are logged until either the interface window becomes full or they encounter a mispredicted branch. At this point the *simulator* is invoked which updates its state based on that information. They report slowdowns of 1300x compared to native execution. Out of this, 130x is introduced by MINT instrumentation and 10x by their interface-window based simulator model. They also simulate a multiprocessor configuration where the slow down because of the interface window is in the high twenties. To keep the model simple, they ignore the cache pollution due to wrong path executions.

2.3.3.2 Checkpointing

To gain the time spent in doing functional simulation and also to accelerate detailed simulation, scientists use *checkpointing*. Checkpointing consists of dumping program and architecture state at certain points to a file during detailed execution. Next time,

to start simulation from one of those points, instead of executing the program until that point, a simulator can directly load the processor state from the checkpointed file and start executing. Disk space requirements to store the simulator state for many points can grow quite large. Schnarr and Larus [89] introduce speculative direct-execution as well as memoization in their paper. Memoization consists of storing a compressed architecture state and the following detailed simulator actions. Next time the same state is encountered, instead of playing the instructions in detailed simulation mode, the stored actions are replayed. This results in speedup for those actions. They report that storing simulator states for one SPEC CPU95 program could require up to 900 megabytes. Barr *et al.* [9] recommend using the *memory timestamp record* which, for every block of cache in a multiprocessor system, stores the last time each processor accessed it. They propose to store this information in a checkpoint and then reconstruct the cache and directory state from it when loading the checkpoint for a detailed simulation. To reduce the size of the stored checkpoint, Biesbrouck *et al.* [103] recommend using only the words of memory that will be needed (*touched memory image*) or only the values for the loads (*load value sequence*) that will be executed next. To avoid repeating checkpointing for different cache sizes, they propose the *memory hierarchy state*, i.e., storing the state of a large cache and then constructing the smaller caches from it.

2.3.3.3 Parallel Simulation

Checkpointing has the added advantage that once we have multiple checkpoints, we can distribute them on different processors to do simulations in parallel. Lauterbach [65] proposes using checkpoints to distribute traces over different machines for parallel execution. Eeckhout and Bosschere [27] also recommend distributing sampled traces over different machines for simulation though, instead of checkpoints, they use detailed simulation to restore cache states. Reinhardt *et al.* [88] describe the

simulation of a parallel shared memory machine on a parallel message passing machine. They distribute the target's nodes on the host machine and the host machine directly executes all the instructions that hit in the target's cache. Only instructions that miss in the target's cache are passed on to target simulator so that it can use latency and sharing information to update its state.

Using a modular simulation environment [101], Penry *et al.* [80] propose dividing the execution of a CMP simulator into threads and executing these threads, in parallel, on a multi-processor server. They report a maximum speed-up of 7.63 when simulating a 8-core CMP, parallelized in 4 threads, running on a 4-processor server. As an alternative, they propose replacing the processor components of the simulator with their counterparts implemented in hardware on an FPGA. Integrating the hardware implemented processor cores in their simulator model, they were able to achieve a speed-up of 5.82 for an 8-way CMP model with *perfect* caches and 1.31 for an 8-way CMP with *non-perfect* caches.

2.3.3.4 Sampling

In the field of statistics [87], sampling is used to determine the characteristics of a large population by observing those characteristics for a smaller subset. The basic assumption is that a well chosen subset sample correctly reflects the population's characteristics. Sampling is popular because it reduces the cost and the effort to collect population data.

Due to long simulation times for detailed cycle-by-cycle execution of benchmark programs, sampling has attracted the attention of architecture researchers as an attractive alternative to full program simulation. The idea is that if we consider a program execution as a population of instructions or basic blocks, observing the performance of a subset of this population should be able to give us an accurate picture of the performance of the whole population, i.e., the program. Relying

on the well established and well tested theoretical background developed by the statisticians, architects simulate most of the program in functional fast forward mode and only a part of it in detailed performance measurement mode to get insights into program performance. Running most of the program in fast functional mode greatly reduces the total simulation time as it is at least an order of magnitude faster than the detailed mode. Sampling can be applied to simulation both in trace driven mode as well as in execution driven mode.

Earlier sampling studies [96, 77] tried to simulate *one* continuous chunk of the program instruction sequence and used its performance as a replacement for that of the program. It was found that different portions of the program could have significantly different performances, especially the initialization phase at the beginning of the program is quite different in performance from the rest of the program. This was demonstrated by [16].

Laha *et al.* [59] provide a method to sample the memory address trace of the program to determine the mean miss rate and the distribution of the miss rate for processor caches. With a sample size of 35, by simulating only 7% of the trace, they were able to show that the cache miss rate distribution of the sample matched that of the program trace.

Conte *et al.* [16] use trace sampling to calculate the IPC for SPEC CPU95 programs. They decompose the source of error into *sampling* and *non-sampling* bias. Sampling bias is inherent in the sampling process and can be indicated by the variance of the sample. It can be alleviated by increasing the number of samples. They recommend the use of warm-up to reduce non-sampling bias. Poursepanj [84] uses trace driven sampling methodology to model the PowerPC 603 processor. Martonosi *et al.* [71] study the effect of the number of samples, warm-up and sample length on the accuracy of sampled measurements. Lauterbach [65] collects randomly distributed trace samples and recommends executing them in parallel to

reduce simulation time.

Sampling can be divided into two types based on how the samples are selected: *random* and *representative* sampling.

2.3.3.4.a Representative Sampling

Representative sampling consists of grouping the elements of a population based on their similarities and then choosing a representative sample for each of these groups. Skadron *et al.* [94] divide their programs into intervals of 1 million instructions each. They then measure the branch misprediction rates for each interval. They note that program behaviour shows *phases* over time. They then select a sample of 50 million instructions which is representative of the general program behaviour. They recommend skipping the initial phases of the program when selecting this window. Lafage and Seznec [58] also divide their program into intervals of 1 million instructions but instead of branch misprediction rates they use statistics about temporal and spatial locality of the memory references to classify their intervals. They use hierarchical classification to group their intervals together and then for each group they choose a representative to simulate based on its closeness, via euclidean distance, to the center of the group. They report simulating on average 1% of the trace and having an absolute error less than 10% for cache miss rates. Dubey and Nair [25] propose *profile driven* sample generation. They do a first execution of the program and save the *profile* of the application in terms of the frequency of basic blocks. They instrument the program binary for a second execution in such a way that the execution count of each basic block is reduced by an acceleration factor. This way, the sequence of the basic blocks executed is the same as in the original trace. The number of executions of each basic block is reduced proportionally to shorten the trace. They report a 50% improvement in error when compared to existing techniques.

In [90], Sherwood and Calder plot the behaviour of SPEC CPU95 programs

as a variation of time. They notice that program behaviour repeats cyclically in terms of IPC, branch prediction, and cache performance. They show in [91] that this repetition of program behaviour is linked to repetition in program code. They propose to create basic block frequency vectors (BBVs) for program intervals of 100 million instructions and then find the interval whose BBV is closest to the BBV of the whole program. They then select that interval as the representative of the whole program for simulation.

In [1], Annavaram *et al.* used sampled hardware program counters to make Extended Instruction Pointer Vectors (EIPVs) and try to correlate them to program performance. They found that while for some programs the EIPVs captured correctly the behaviour of the program, for others it was not the case. Lau *et al.* [61] retorted that while sampled EIPVs could have a fuzzy relationship with performance, full code signatures, like BBVs, do show strong correlation to code and can help predict program performance.

In SimPoint [92], instead of choosing one representative, they group the generated BBVs into multiple clusters by using the k-means clustering algorithm. Then they find out the representative of each group and then use all these slices of program as representative of the whole program. They term these groups containing intervals with similar characteristics *phases*. In [93] they propose a mechanism to detect and predict these phases. They show that phase prediction could be used to help value prediction and dynamic adaptation of processor width and data cache. In [62], Lau *et al.* use and compare structures other than BBVs to compare phase classifications. They use loop branches, procedures, opcodes, register usage, and memory address information to characterize program intervals. They found that BBVs performed the best. In [60], they try variable length intervals so that they might be better aligned with basic block boundaries and code structure.

In [68], Liu and Huang show that different invocations of subroutines during

the execution of a program show low CoV (Coefficient of Variation) percentages for metrics like CPI, basic block size, branch prediction, memory references, and L1 cache hit rate. They partition the static code of the program into reasonable sized subroutines. Then they try to simulate a dynamic instance of each of these code sections. They propose two methods to do simulation. One with preprocessing where they run the application to determine population size and then choose a *systematic sampling* rate and they run it again to create checkpoints for selected samples. In the second case they avoid the preprocessing phase and sample without prior knowledge of the incoming code. To capture variation between successive invocations they try to spread out the sampled invocations through out the execution. At the end of their paper they do suggest an online method which characterizes code intervals and uses that characterization for selective sampling on the go, but they do not implement it.

Phase Classification and Prediction. In order to select a few representative portions of the program to simulate, we need to group the program portions into groups based on their similarity. Such techniques can be qualified by measuring the homogeneity in the groups thus generated and how different they are from one another. Dhodapkar *et al.* [22] compare program phase characterization techniques such as *working set signatures*, *basic block vectors*, and *conditional branch counters* in terms of phase classification accuracy, average phase length, and phase stability. They state that phase detection when using basic block vectors performs better than the other two techniques. Although the instruction working set technique generally gives longer phases than the other two, there's less stability in those phases as compared to phases detected by BBVs. The conditional branch counters are slightly less accurate than the other two techniques but are much easier to implement. They note however that all the techniques agree with each other 85% of the time.

Lau *et al.* measure the intraphase homogeneity by using CoV (Coefficient of Vari-

ation) for each phase to qualify the efficacy of their phase classification technique. Kodakara *et al.* [56] observe that the CoV based classification can be inconsistent under certain conditions. They propose the CIM (confidence interval of estimated mean) as a better alternative to estimate the homogeneity of the phases.

Lau *et al.* [64] classify the transitional period between stable program phases as a separate phase. They take up the phase predictor from [93] and use it to predict phase change and the ID of the next phase. They add confidence counters to these predictors to improve prediction. They also use these predictors to predict the length of the next-phase burst. In [105], Vandeputte *et al.* examine and compare existing phase-prediction techniques. They compare the *last value predictor*, *N-level burst predictor* and *N-level RLE predictor*. They found that simple predictors, like, last-value predictor, perform poorly for programs with frequently changing behaviour when compared to more sophisticated predictors. Between the N-level burst predictor (which keeps the IDs of the last N distinct phases) and the N-level RLE predictor (which keeps the burst lengths in addition to the IDs of the last N predictors), they found the N-level burst predictor to perform slightly better for limited hardware budgets. They also noted that addition of confidence counters and conditional update to these predictors further decreased the misprediction rate. Duesterwald *et al.* [26] characterize the program behaviour with respect to metrics like instruction mix, branch prediction accuracy, data cache miss rate, and instructions per cycle, by sampling hardware counters at regular intervals. They show that program behaviour repeats in a synchronized manner across different metrics. They predict the future values for different metrics by looking at their histories. Using different sorts of predictors (last value, mode, median, exponentially weighted moving average (EWMA), fixed-size history and run-length encoded history), they found that for programs showing a high degree of variability, table-based predictors like (fixed-size history and run-length encoded history predictors) perform better

than the statistical ones. They propose to use this prediction of program metrics to proactively activate hardware/software optimizations to take advantage of program phase behaviour.

Falcón *et al.* [33] propose Dynamic Sampling using virtual machines to fast forward the functional portions of the program. They use the AMD's SimNow^{TM} simulator to execute the functional part on the host machine and PTLsim [118] to simulate the timing information for samples taken. Leveraging the correlation between program performance and code metrics [62], the functional simulator in the virtual machine monitors certain parameters for significant change to detect the change in program phases. These monitored parameters include *code cache invalidations*, *code exceptions*, and *I/O operations*. During functional execution, if the virtual machine detects that change in the monitored parameter, between two consecutive intervals, has exceeded a certain threshold, it proceeds to take a sample. In this mode, the virtual machine generates events which are consumed by PTLsim [118] to generate timing information. Once the sampled interval is finished, the machine reverts back to the functional mode. These sampled measurements are used to estimate the full-program performance at the end. Executing the functional *(major)* portion of the program in a virtual machine on the host processor enhances the speed of the simulation. They compare their speedup and accuracy to SimPoint [92] and SMARTS [110]. They note that though SMARTS is the most accurate technique on average, its functional warm-up severely limits the speedup. SimPoint achieves good speedup only when we do not count the initial profiling analysis. Dynamic Sampling provides good accuracy and speedup in general but, at the same time, they admit that the error and speedup depend on the monitored parameter and the phase-detection threshold. Monitoring *code exceptions* is less accurate than keeping track of the *code cache invalidations* and *I/O operations*. The worst case error for a benchmark was reported to be 8%, $> 10\%$, and $> 20\%$ for SMARTS,

Dynamic Sampling, and SimPoint respectively. This same sampling mechanism has been integrated in HP's COTSon simulation infrastructure [2].

In [44], Isci and Martonosi study the use of performance monitoring counters and basic block vectors for phase characterization of processor power behaviour. Using binary instrumentation of programs, they sample a PC address every 1 million instructions and after every 100 million instructions they get a 32-dimension hashed BBV. Similarly they use hardware counters to count 15 hardware events such as, L1 and L2 access rates, bus utilizations, etc. Every 100 million instructions they obtain a vector detailing the statistics for the previous interval. They use clustering to classify both the BBVs and PCVs (performance counter vectors) into phases and compared the phase classification to the power phases. They found that PCV based phases outperformed the BBV based phases every time (33% better classification on average). They trace this to two phenomena that the BBV code signatures fail to capture: *Operand Dependent Behaviour*, where the same code execution, when used with different arguments, result in different data locality behaviour and, hence, different utilization of hardware resources (cache, buses, etc.) and different power phases, and *Effectively Same Execution*, where different portions of executed code, which the BBV classification mechanism qualifies as different phases, result in same power signatures. They give examples of these two phenomena and show that they are correctly identified by the hardware performance counter signatures. They recommend using the PCV based phase information to effectively scale processor voltages in view of the power consumption.

2.3.3.4.b Random Sampling

An alternative to representative sampling is statistical random sampling. This method avoids characterizing the population beforehand and, instead, picks samples from the population in a random fashion. Random selection assumes that since

the selection process is free from any bias, the sampled units will reflect the characteristics of the population, i.e., there will be more samples from more frequently occurring portions and vice versa.

In [16] authors advocate the use of random sampling for program trace simulation to avoid simulating long traces. They try to devise a methodology to limit the error in sampled simulation by stressing on reducing both the sampling and non-sampling bias. Lauterbach [65] also uses random sampling of program instruction traces to measure program performance.

SMARTS [110] uses a variant of random sampling called *systematic sampling*. Systematic sampling selects samples to simulate *at regular intervals* from the execution stream of the program. Though the regularity of occurrence is predictable in systematic sampling, choosing the start point randomly makes each interval of the program have an equal probability of selection. A potential problem with systematic sampling is that if the program contains repeating behaviour whose period is a multiple of sampling frequency, then sample selection will be biased towards one kind of behaviour. Wunderlich *et al.* verify that this is not the case by measuring the homogeneity in the program via intraclass correlation coefficient. Using a sample size of 1000 instructions SMARTS reports achieving an average CPI error of less than 1%.

Wenisch *et al.* propose SimFlex [107], a statistical framework, to systematically sample the Transaction Processing Workloads on multiprocessors. Due to the high variation in the transaction completion rate, which makes them simulate longer intervals, they use the retired *user mode instructions per cycle* (U-IPC) as their performance measuring criterion. U-IPC shows much less variation than the transaction completion rate. They first do an initial brief sampling to measure the variation in performance; this is used to calculate the number of measurements needed to achieve a specified confidence. As a next step they measure the detailed warming length

needed to remove the cold state bias from their measurements. The third step is the actual simulation phase.

In [31], Ekman and Stenstrom use matched-pair comparison to measure and calculate the variation in differences between pairs of samples selected by running a program on a baseline architecture as well as the architecture to be tested. The assumption is that the variation in the difference of pair of IPCs on two architectures is much lower than the variation in IPC over one architecture.

Azimi *et al.* [7] use statistical sampling of hardware performance counters to estimate different events affecting the program's performance. Due to the limited number of counters, they use multiplexing of these counters to model more events. They also propose a heuristic mechanism to attribute these events to stall cycles in order to identify performance bottlenecks. They show that multiplexed statistical counters can estimate the counts of hardware events within 15% of the real counts with an implementation overhead of 2%. Using their heuristics they show that most of the stalls result from data cache misses. They propose to use these stall sources identified by the sampled counters to provide hints to guide the runtime optimizer towards useful optimizations.

In [111], Wunderlich *et al.* try to see the effect of combining stratified sampling with simple random sampling. In their stratified random sampling they use two methods to classify the program behaviour into strata, basic block vectors (BBVs) and instructions per cycles (IPC). They observe that while program classification into BBV based strata can reduce the simulated instructions by half, the additional complexity of profiling and clustering diminishes the gains. The second approach was to create the IPC strata for a program by detailed simulation on an architecture and use this information to reduce the number of randomly simulated instructions on another architecture. They found that this approach may work only on architectures very similar to the baseline.

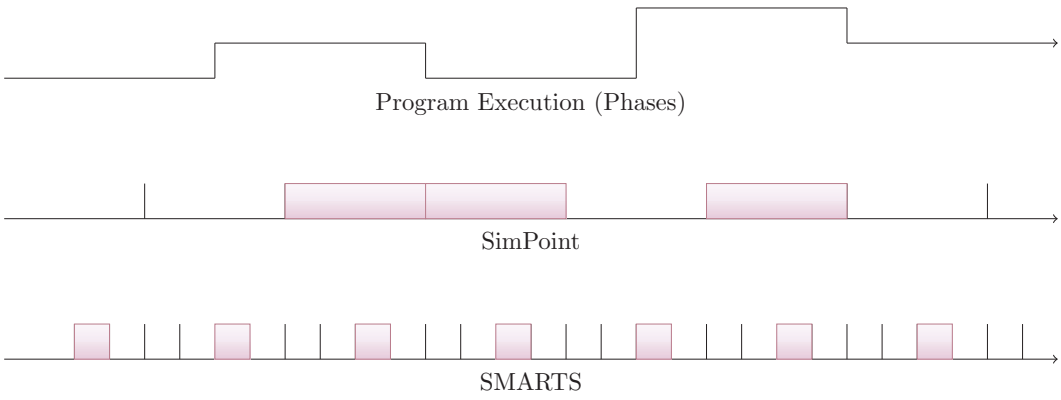


Figure 2.1: *SimPoint and SMARTS mechanisms.*

2.3.3.4.c Comparison of Techniques

The qualitative and quantative comparison of different simulation acceleration techniques help put things into perspective. In [115] Yi *et al.* compare existing simulation techniques like, truncated execution, input set reduction, random (SMARTS) and representative (SimPoint) sampling, etc., based on a number of criteria. They found the sampling based techniques to perform better than truncated execution and reduced input sets on the accuracy front. They are also more configuration independent than their counterparts. Secondly, SMARTS is slightly more accurate than SimPoint but SimPoint provides better speedup. On the ease of usability (complexity to implement) side, SMARTS is relatively complicated to implement as it exposes the user to the internal details of the simulator (for functional warm-up). Nookala [6] compares MinneSPEC and SMARTS performance for microarchitecture aware floorplanning to get an optimal block layout for the chip. They found both to perform almost identically for this particular problem and suggest the merger of the two. Figure 2.1 shows how SimPoint and SMARTS sample program phases.

2.3.3.4.d Multi-threaded Sampling

New computing systems increasingly employ multi-core processors (even in embedded systems) running multi-threaded applications. So far simulation acceleration techniques have been focused on improving the simulation times for single-threaded applications. This surge in the use of multi-threaded applications has prompted attempts to accelerate the simulations of such applications.

Van Biesbrouck *et al.* [102] extend the SimPoint [92] approach to sampling Simultaneous Multithreading (SMT) processors. Using detailed simulation of multiple programs on a multi-context processor they show that though the resource contention between applications affects their time-varying behaviour, they still retain their repetitiveness of performance. They exploit this *collective* repeating behaviour to reduce simulation times. They create the *Phase-ID traces* for each program running in isolation using SimPoint. When more than one programs run together, their phases overlap in execution. These combinations of phases occurring together are termed as *co-phases*. The basic idea is the same that once the performance for a co-phase has been measured, the next occurrence will exhibit similar behaviour and need not be simulated. The co-phase IDs together with their performance measures are stored in a *Co-Phase Matrix*. Using the current co-phase ID and the Phase-ID trace of each program, they can determine how many instructions they can skip before they encounter a phase change in a thread that results in a new co-phase. They fastforward until they arrive at a new co-phase with no entry in the Co-Phase matrix; they simulate it in detailed mode and store its performance in the matrix for future use. They test the mechanism on a two context SMT processor and report an average error of 4%.

CoGS-Sim [49] combines the co-phase [102] and PGSS [54] simulation techniques to provide an online sampling strategy for multi-context SMTs. It proposes to keep track of the co-phases in an online scheme which hashes the branch addresses into

fixed size BBVs. Using *pivot clustering* it classifies each new co-phase into an existing or new cluster. Additionally, upon detecting a co-phase change, it tries to obtain a detailed simulation sample for the new co-phase. Using the SPEC CPU2000 benchmarks on a two-way SMT processor it reports an average percentage error of 15-20%. They cite lack of warm-up and statistical error as possible candidates for sources of errors. They recommend a thorough exploration of the parameter space to further optimize the technique.

Tawk *et al.* [98, 100, 99] use representative sampling to accelerate the simulation of multi-core system on chip (MPSoC) platforms. They use SimPoint to extract a phase profile of the simulated programs in a pre-simulation functional run. Simulating on a multiprocessor configuration they build strings of phases, being executed in parallel for each process, and store them in a cluster table. For each unique phase string combination, they try to sample a performance measurement. When the combination of phases to be executed by the parallel processes already has a performance reading in the cluster table, they skip its simulation via fastforwarding to save time. In *Adaptive Sampling* [98], when a process reaches at the end of its phase interval, it tries to estimate the time (cycles) remaining for other processes to finish their intervals. If this time falls within its acceptable threshold, it waits for them to finish their phases and then a performance measurement is taken for this combination of phases. This *wait* lowers the IPC for that phase of the waiting process and introduces a source of error. In multi-granularity sampling [100], they correct this error by adjusting the IPC for that phase using empirically determined average IPC values. In the same work, they also combine multiple consecutive intervals into variable length samples in order to reduce the number of unique phase combinations.

2.3.3.4.e Warm-up

A sample's performance (IPC) would depend on the performance of the underlying components (caches, branch predictors, prefetchers) which in turn depends upon the data present in these structures. This, together, constitute the state of the machine. One issue with sampled simulation is having the correct architecture state before the sampled interval. For example, consider starting the simulation of a sample with empty (or having stale data) structures. When the code in that portion will try to access these structures, it would incur cache-miss penalties and branch misprediction penalties. This would result in an *inflation* of the CPI. If this code had been executed in the course of regular execution of the program, these structures would have had updated data and would not have incurred these (false) penalties. This brings us to the problem of removing cold start bias [16] or warm-up.

Simple approaches. There have been many approaches when restoring the state of the simulator architecture for sampling. *Cold start miss* considers that all data in the cache is invalid or an empty cache and returns a miss for the first access to every block of the cache. *Cold start hit* assumes the opposite and makes every first access to a cache block a hit. It is clear that they both can result in an over or under estimation of the CPI respectively depending on the program nature. Tawk *et al.* [100] and Biesbrouck *et al.* [102] use the *cold start hit* warm-up for their sampling experiments. *Stale state* advocates using data left over in microarchitectural structures from the previously simulated-in-detail interval. This can have mixed results depending on the distance between data reuse in a program.

Crowley [19] cites sampling traces for Windows NT applications on Intel x86 platform and using 4 methods to remove cold-start bias: *cold*, where there's no warm-up, *half*, where first half of each sample is used for warm-up, *stitch*, where the state at the end of previous sample is used at the start of current one, and *INITMR*, which estimates the miss ratio for references in the sample. He found that for large

caches none of these techniques performed well.

Functional warm-up. SMARTS [110] proposes using *functional warm-up*, i.e., updating the states of macroarchitectural structures between two samples during the fast forward phase. This lets them use just a few instructions to warm-up the microarchitectural structures before the sample. As noted by Yi [115] this shifts the onus of modifying the functional simulator on the end user. Skadron *et al.* [94] also use functional warm-up updating only the "caches, branch predictor, and architectural state". In [68], the authors propose using either functional warm-up or checkpointing to attain correct architecture state before sampling.

Detailed warm-up. Detailed simulation has been suggested as an alternative for warm-up. A portion of the program before the sample is simulated in detail with the hope that most of the *false* misses would occur during this phase and would result in correct data in architectural structures. Some studies suggest using a fixed number of instructions in detailed mode to do the warm-up. In such situations the problem is that a chosen fixed warm-up size may be correct for one configuration of architecture and not for another. Haskins and Skadron [47], in presenting MRRL, count the number of completed instructions between consecutive references to each unique memory location. They select a warm-up length that covers the reuse for N% of the references. This way they are able to shorten the length of detailed simulation needed to achieve a N% warm-up for that sample. BLRL [28] is similar to MRRL except in stead of counting the reuse latencies of all memory references, they count them for those memory references whose consecutive accesses cross the sample boundary. This results in shortening of the warm-up length. NSL-BLRL [104] leverages the warm-up length reduction by BLRL to reduce the checkpoint size by storing only information about the memory references that BLRL thought were important. Self-Monitored Adaptive cache warm-up (SMA) [69] monitors the accesses to cache blocks by adding a bit to it. The first access to a block is considered

a cold one but it sets the bit. All next accesses to the same block are deemed warm. It signals the completion of warm-up in two fashions: either a certain percentage of cache blocks are accessed, or the percentage of warm memory accesses exceeds a certain threshold.

Budgeted Region Sampling [35] is a technique which addresses the issues of both sampling and warm-up. The authors divide the program intervals into *regions* based on the differences in basic block reuse distance. They then use the IDDCA [36] algorithm to cluster these regions. Then they divide a fixed detailed simulation budget between these regions based on their relative weights. Within each region, the allocated budget is divided between warm-up and sampling based on the length of the region.

Kihm and Connors [53] try to use statistical sampling to determine the performance of multithreaded architectures. During the fast forward phase they fast forward each thread based on the value of its IPC seen in the last detailed simulation. This *proportional fast forwarding* tries to recreate the same co-phase overlappings as would occur in a complete detailed simulation. To recreate the system state before each sample, they experiment with *Monte Carlo warming*. The problem with warm-up of large shared structures is that we do not know how much of them was affected by each thread. This depends on the time-based interleaving of the instructions of the threads. Monte Carlo warming interleaves the memory and branch instructions of the threads randomly based on their last seen IPCs. This has the effect of maintaining cache affinities for the threads proportional to their executed instructions. They also found that doing a functional warm-up improved the results while costing a 16% decline in simulation speed.

Checkpointing. Checkpointing has also been recommended as one of the methods to restore the states of caches and branch predictors, etc., before starting the simulation. Checkpointing requires a prior simulation of the program and knowledge

of the position of the sample. During that prior execution, just before the start of each sample, the simulator dumps information related to the state of the architecture in a checkpoint file. When simulating the samples, information in these checkpointed files is used to reconstruct the state of the architectural structures. Lauterbach [65] proposes using checkpoints to load the simulator state when sampling traces. An advantage of checkpointing is that since a sample needs only *its* checkpoint to simulate, samples can be distributed on different machines with their checkpoints to attain parallel execution of them. The inconvenience is that storage space needed to store multiple checkpoints for multiple programs for multiple architecture configurations may become excessive. Wenisch *et al.* provide a checkpointing mechanism for the SMARTS [108] methodology. Tools exist [78] to instrument execution files to generate checkpoints at specified points. Biesbrouck *et al.* [103] propose techniques to reduce the size of stored checkpoints without affecting their capability to remove cold start bias.

Functional warming [110] is done for almost 99% of the program execution and thus any slowness in its mechanism can slow down the speed of the simulations. In addition, functional warming depends on the length of the simulated program and with time the lengths of the benchmark programs continue to increase; SPEC CPU2006 programs are on average 10x larger than 2000 ones. Wenisch *et al.* [109] propose a checkpointing approach to replace the functional warming. In order to keep the size of stored checkpoints small, they propose only to store the state for structures which have a long history of warm-up, i.e., caches, branch predictors, etc. The small structures are updated with detailed warming. Storing the full state for large cache-like structures can take large disk spaces. In order to counter this problem, for each checkpoint, they propose to use the memory reference analysis [47] and store only the information that will be accessed in the following sample. Dispensing with the information not pertinent to the measured sample reduces the

checkpoint size by 2-3 orders of magnitude as they use very small measurement intervals. They try to model the effect of cache pollution by speculative execution by maintaining branch predictor outcomes when creating their *live points*. By creating independent checkpoints, they also exploit the advantage of parallel execution of samples and report an average runtime of 91 seconds and a total checkpoint size of 12GB for the SPEC CPU2000 benchmarks.

SimFlex [107], which builds upon Simics [70], uses Simics' checkpointing mechanism and live points' approach to create *flex points*. They create a checkpointing library to help simulate online transaction processing (OLTP) and web server workloads on a multiprocessor.

2.3.3.4.f Combining Sampling and Warm-up

While most studies consider the Sampling and Warm-up issues orthogonal and, thus, tackle them separately, there have been attempts to graft them together to achieve an inclusive methodology. Since we are looking for a one-pass method that does Sampling and Warm-up on the go, these attempts deserve a mention.

In SMARTS [110], Wunderlich *et al.* integrates the warm-up with their sampling strategy by proposing to update large microarchitectural structures, i.e., caches, branch predictors, etc., during the functional phase of the simulation.

Pereira *et al.* [81] and Kihm *et al.* [54] also combine the sampling and warm-up mechanisms at runtime. Online SimPoint [81] uses the phase tracking mechanism of [93] to track and predict program phases online. Predicting phases makes it possible for them to anticipate the occurrence of a phase and prepare the architectural state for sampling before this occurrence. They maintain a moving queue of a certain number of memory and branch events that occur during the functional fast forwarding phase and then use this queue to warm up the simulator state before sampling.

PGSS [54], unlike Online SimPoint [81] which predicts phases, relies on just phase change detection. Once it has detected a new phase, it proceeds to take a sample for that phase before the phase is over. It relies on functional warm-up, as proposed by SMARTS [110], between two samples.

2.4 Conclusion

In this chapter we discussed the current state of the art in the field of architecture simulation. The amount of activity going on and the amount of resources being spent on it shows the importance of making design choices early in the processor fabrication process and the need to do it fast. Though it is equally important to reduce the number of design space parameters and choose the correct benchmark-input pairs, we decided to focus on the third aspect of design space exploration which is reducing the individual simulation times. Even within that we focus on Sampling because it is the technique that has shown the most potential. It not only reduces the simulation execution times (often by orders of magnitude) but also the performances reported are very close to the real ones (sampling errors are usually in lower single digits).

Having studied state of the art on architecture simulation and sampling techniques, we found there can be improvements in certain areas: that not all state of the art sampling techniques are transparent to architecture and software modifications, that the best warm-up methodologies are not adaptable to sampling techniques, and that the efforts to integrate warm-up and sampling into an online, one shot, mechanism have their limitations. All these issues are important in their own right. While previous attempts to address these issues have focused on them mostly individually, we attempt to provide a methodology which incorporates them all. In the following chapters we present sampling approaches which aim at tackling these issues.

Chapter 3

Transparent Representative Sampling

3.1 Introduction

Sampling relies on the assumption that most of program execution is a repetition of parts of code so it is not necessary to performance simulate the entire execution of the program. It suffices to simulate in detail representative portions of the program code only once and from the performance of these code segments that of the whole program can be constructed. This idea that a program execution comprises repeating *phases* was popularised by SimPoint [92]. While SimPoint [92] *et al.* [81] [54] provide good performance estimates, they ignore other issues which are equally important. Some use a prior phase analysis to determine which portions of the program to simulate in detail making the technique cumbersome to use in case of frequent software modifications. At the same time, other techniques make too simplistic assumptions about the warm-up of SRAM structures which can have significant effect on performance measurements in case of change in architectures. We propose a holistic approach to all these issues with equal emphasis on usability,

transparency to architecture as well as performance accuracy. This technique, which will find the representative portions of the program and, at the same time, adjust itself to the changes in hardware and software, we call *Transparent Representative Sampling*. The core simulator implementation is called the *Transparent Sampling Engine (TSE)*.

3.2 Repetition of Program Behaviour

Program executions show variable behaviour over time. This behaviour may appear different depending on the scale on which it is measured. The performance fluctuates frantically on small scale but shows a smoother variation when measured on larger scales.

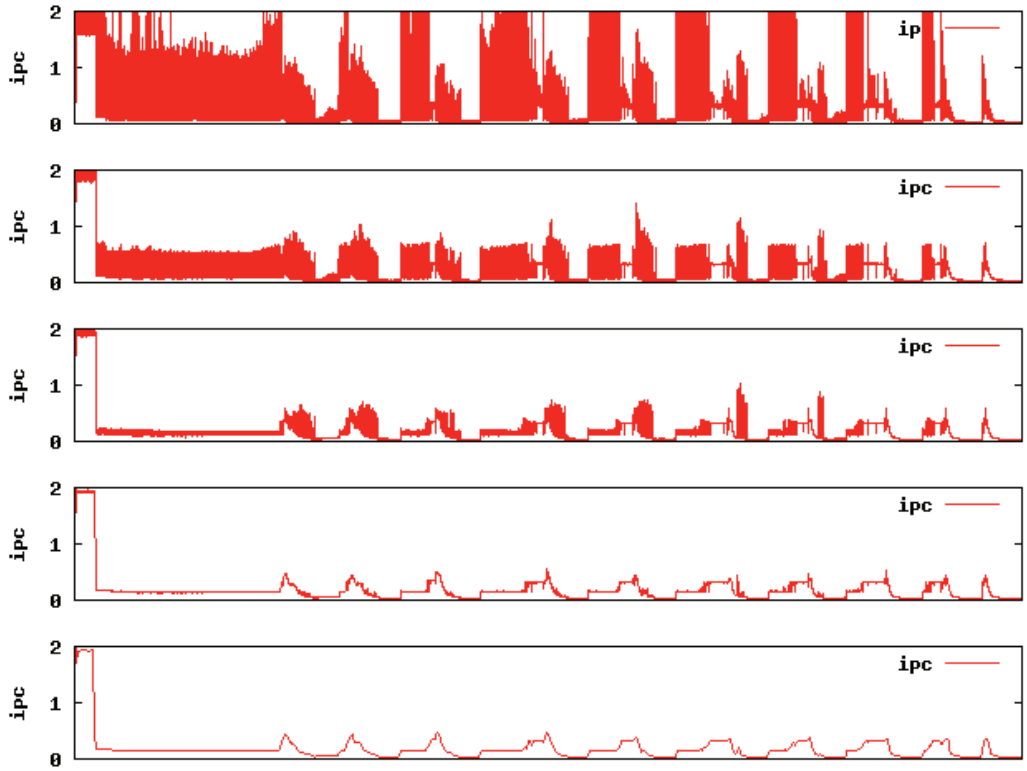


Figure 3.1: *IPC variation for MCF. Interval sizes of 10k(top) to 100 million(bottom).*

Whatever the scale of measurement, it can be observed that a program behaviour consists of repeating patterns of performance. These portions of a program with same or similar metrics are grouped into *phases* [92]. A phase may recur many times during the execution and these occurrences need not be temporally adjacent. While each phase is distinct from others in terms of the metric measured, behaviour within a phase is fairly homogeneous. Furthermore, these repetitive changes can be correlated with different portions of program code. What this means is that executing the same code again at another moment in program execution timeline would result in the same performance measurements. Since a program execution consists of repeatedly executing same portions of code, hence the repetition in performance. This is a key result which helps understand the program behaviour and makes it exploitable. It opens up whole new areas in program optimisation. Once one knows that a certain portion of code is going to repeat in future, one can study its performance and, in reconfigurable architectures, reconfigure the processor to parameters best suited for its performance. Calder *et al.* [93] study the effect of adapting the data cache size and the processor issue width dynamically with program behaviour. They found out that customizing the architecture to executed code can yield better performance and can reduce energy consumption.

This result that a program execution consists of repeating phases is exploitable in architecture simulation as well. If we know that a phase is going to execute multiple times and each execution will be resulting in the same performance behaviour, it is redundant to simulate it each time we encounter it. Simulating it the first time would give us the performance information about its future executions as well. This way simulating each phase only once, not only can we estimate the performance of the whole execution of the program but also we will reduce the simulation execution time. If a program consists of N phases, its performance can be estimated by:

$$CPI = \sum_{i=1}^{N} (CPI_i \times m_i)$$

where CPI_i is the performance measured by the single simulation of ith phase and m_i is the fraction its occurrence constitutes of the whole program.

Once we reach the above conclusion, it remains to classify the program execution into phases and then selecting an instance of each phase for simulation. This is detailed in the next section.

3.3 Phases and Code Signatures

The changes in a program phase behaviour are obvious when simulating the whole program in detail; we can see a clear change in the performance metric. We need a mechanism which helps detect these phases without launching the detailed simulation. There have been attempts in the past to categorize the program phases using methods other than full detailed simulation. Dhodapkar *et al.* [23] show that program phase changes are related to the instruction working sets. They concluded that changes in program behaviour tend to coincide with the changes in working sets. Balasubramonian *et al.* [8] detect changes in program phases by keeping track of conditional branch counters for each interval in program execution. When the difference between the conditional branch counters of two consecutive intervals exceeds a certain threshold, they signal a phase change. Sherwood *et al.* [91] advocate the use of *Basic Block Vector (BBV)* analysis to detect program phases. Dhodapkar and Smith [22] did a study comparing the three approaches and found that the Basic Block Vector approach performed better than the other two when detecting the phases. In [63], Lau *et al.* compare different structures for phase classification and concur that BBVs are one of the most accurate elements to capture the phase behaviour of the programs. Based on these results we selected the changes in BBVs as our criterion of choice when detecting the phases.

3.4 Basic Blocks and Basic Block Vectors

A *Basic Block* is a portion of code with one entry and one exit only. Once execution enters a basic block, it is sure to execute the whole of the basic block before it can exit it. These are the smallest units of code guaranteed to execute in their entirety. Basic block boundaries are marked by control statements. Listing 3.1 shows a simple while loop in a high-level language, such as C, and the Listing 3.2 shows the corresponding MIPS assembly code. If we see the code in Listing 3.2, we can see that there is one point of entry in this block; this is the instruction labeled "loop" on line 100. However, there are two exit points, namely: the branch statement on line 103 where the program exits the loop and the jump statement on line 105 which jumps backward to the beginning of the loop. Therefore the program execution can follow two paths in this piece of code: it can either do from line 100-105 in an ordinary iteration of the loop or it can execute the 100-103 instructions in the last iteration before exiting. There are two basic blocks which are separated by the branch instruction in the middle, i.e., line 100-103 and line 104-105.

Listing 3.1: Basic Block Example

```
while ( A[i] == k )
    i++;
```

Listing 3.2: Basic Block Example

```
100 Loop: sll $t1, $s3, 2
          add $t1, $t1, $s6
102       lw $t0, 0($t1)
          bne $t0, $s5, Exit
104       add $s3, $s3, 1
          j Loop
106 Exit:
```

A *Basic Block Vector (BBV)* is an array whose elements contain the frequency of basic blocks. The length of the array equals the number of total basic blocks and each element corresponds to one of them. The value contained in a certain element will indicate how many times that particular basic block has been seen during execution. Thus these BBVs form the code signatures for different intervals of the program. Changes in these BBV code signatures are used to detect changes in program phases.

BBVs for program intervals can be collected by doing a first pass using *functional only* simulation which is much faster than *detailed* simulation. SimPoint [92] uses this approach. They divide the execution of the program in *intervals* of equal size and analyze the code for each interval in terms of the basic block vectors. They then group the intervals with *similar* BBVs together into clusters. The idea is that these clusters are representatives of program phases and intervals within a cluster will show similar performance because of their similarity in BBVs. To test the difference

between two basic block vectors they tested both the Euclidean and Manhattan distance and found the latter to perform better in BBV classification.

$$EuclideanDistance = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}.$$

$$ManhattanDistance = \sum_{i=1}^{n}|a_i - b_i|.$$

They employ this distance measure in the *k-means* [79] algorithm to group together intervals whose BBVs are close together. K-means is an off-line iterative algorithm. It needs access to all the BBVs before it can begin the clustering process. As mentioned earlier in the introduction, due to our usability and transparency to hardware and software changes requirements, we want a one pass simulation mechanism and would not have the luxury to access all the BBVs before the end of the execution. Therefore, we desire a scheme that performs clustering on the fly. So we opt for a simpler scheme which, based on a distance threshold, decides whether a BBV interval should go into one of the existing clusters or a new cluster should be created for it. At the end of each interval we obtain the BBV for that interval. We compare this BBV to the centers (BBVs) of all the existing clusters using the Manhattan distance. If the distance between the new BBV and the BBVs of the centers of all the existing clusters is superior to our threshold, we suppose that the performance for this BBV is significantly different from all the other clusters and create a new cluster for it with the new BBV as its center. Otherwise, we merge this BBV in the cluster closest to it by way of the Manhattan distance and update the latter's center. The new center for this cluster is a BBV which averages all the BBVs comprising this cluster including the recently added one.

It is important to carefully select the value for this BBV classification threshold because it will directly control the number of clusters/phases formed and thus effect

the simulation time. Since ideally we would be looking for a representative for each cluster, more number of clusters would mean that we spend more time looking for their representatives in detailed simulation mode. At the lowest threshold, we'll get almost as many clusters as the intervals, except the exactly identical ones, and end up simulating the whole program in detailed mode. On the other end, at the highest threshold of 100% all clusters are collapsed into one big cluster completely obfuscating the phase information of the program as no interval would be different from the other. The effect of this BBV distance threshold has been shown in Figure 3.2. The x-axis lists the benchmarks and the y-axis shows the number of clusters formed when we vary the classification threshold as shown in the legend. The result supports the intuition. At the lower threshold of 0.125 (12.5%) we get the highest number of the clusters for each benchmark and the number decreases consistently as we move to higher thresholds. Three benchmarks stand out due to their relatively high number of clusters: `tiffdither`, `tiffmedian`, and `ispell` have 200, 125, and 116 clusters respectively.

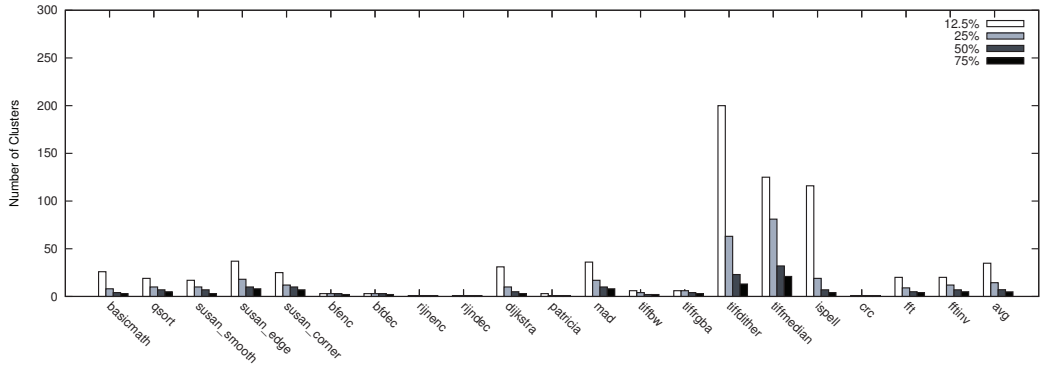


Figure 3.2: *Effect of varying the BBV classification threshold on the number of clusters.*

A normal program can consist of many basic blocks. Sherwood *et al.* [92] found the number of basic blocks for the SPEC2000 benchmarks ranging from 2,756 to 102038. To count the frequency of each basic block individually, the BBV dimensions

have to be quite large. This puts huge strain on the simulator in terms of memory requirements and also slows down the clustering mechanism as it has to calculate distances for each element of the vector many times. Also, with time the size of the programs tends to increase with a corresponding increase in the number of basic blocks. SPEC2006 programs are on average 10 times larger than the SPEC2000 ones [48] and future benchmarks are expected to grow bigger. Therefore using full sized BBVs, especially for on-line phase classification, is quite impractical. For these reasons the original SimPoint [92] article proposes using *Random Projection* [20] to reduce the BBV dimensions. On-line SimPoint [81] and PGSS [54] propose an online version of it. We chose to go with this approach, i.e., online dimension reduction, because low memory and computational requirements are in accordance with our goal of a fast simulation methodology. We use the technique used by PGSS [54] for reducing the dimensions of our Basic Block Vectors. As mentioned earlier, basic blocks are identified by the addresses of their terminating branches. Normally a Basic Block Vector will contain as many dimensions as the number of basic blocks in the program in order to record the occurrence of each. In stead of indexing the basic block by their terminating-branch address, we randomly select a few bits from that address and use those bits to index into the BBV. As an example, selecting 4 bits from the branch address will result in a maximum index of 15 and hence require a Basic Block Vector of size 16. As informed readers may note that this may produce an aliasing-like effect, i.e., merging two basic blocks with completely different *actual* branch addresses and performances. In actual experiments we saw that unless the dimension is reduced to values extremely low, this aliasing effect is tolerable. Figure 3.3 shows the effect on the number of phases detected when we vary the size of our Basic Block Vectors. It can be noted from the figure that when we increase the BBV size from the very low values, the number of phases/clusters increases correspondingly most of the time, revealing that at low dimensions different phases

are forced to mix with each other. However, except a few exceptions, at higher dimensions the number of phases does not increase by much. This is an indication that very high BBV dimensions do not add much in terms of phase information and slightly lesser ones can capture the same behaviour. Notable exceptions are `tiffdither` and `ispell` where the number of clusters continue to increase with the increase in BBV dimension. It should be noted that while we try not to reduce the BBV dimension too much in order to retain phase information, we would also not like it to increase too much. The reason being that larger BBV dimensions require more calculations and result in more number of clusters. A large number of clusters can potentially produce complex cluster sequences which make predicting the cluster occurences hard. This mechanism of predicting the clusters (phases) of a program is the topic of our next section.

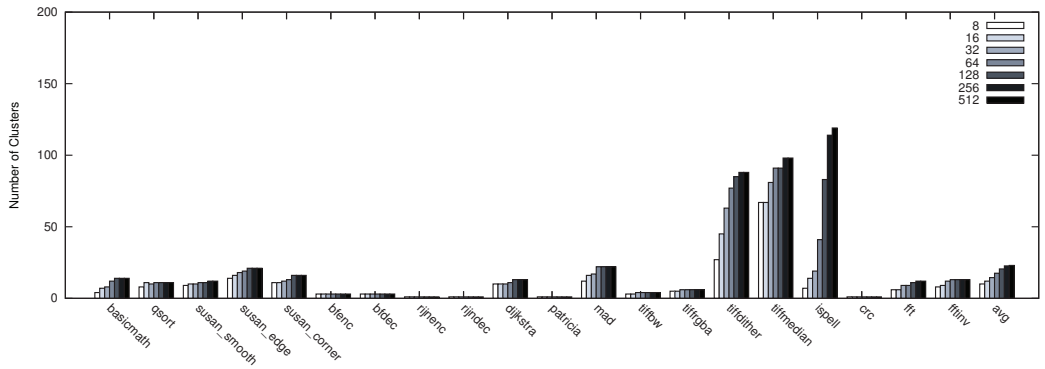


Figure 3.3: *Effect of varying the BBV size on the number of clusters.*

3.5 Phase Prediction

Having the on-line dimension reduction and phase classification technique is not enough. In order to be able to simulate in detail, at least once, each phase of the program, we need information in advance of the phase occurrence. Unfortunately the phase classification technique tells us the identity of the phase only after we

have finished simulating its interval, functionally or otherwise. To tackle this issue, Sherwood *et al.* [93] propose a predictor mechanism which keeps track of different phases being detected and also tries to predict their future occurrences. This way, if predicted correctly, we can anticipate the next occurrence of a phase and start simulating in detail as soon as it starts. They tried different predictors namely *last value predictor*, which assumes the previous phase to repeat, and *N-level RLE (Run Length Encoding) predictor*, which keeps a history of N previous phases and the number of their continuous occurrences. The last value predictor was found to perform poorly in the face of frequently changing phase behaviour. For the RLE predictor, they found that 2-level RLE predictor gives the best trade-off between hardware cost and prediction accuracy.

Perreira *et al.* [81] use the same 2-level RLE predictor. Vandeputte *et al.* [105] did a comparative study of different phase predictors and concluded that though the 2-level burst predictor, which keeps only the IDs of the last 2 phases as opposed to IDs and counts for the RLE one, performs better than the 2-level RLE predictor, the difference is slight.

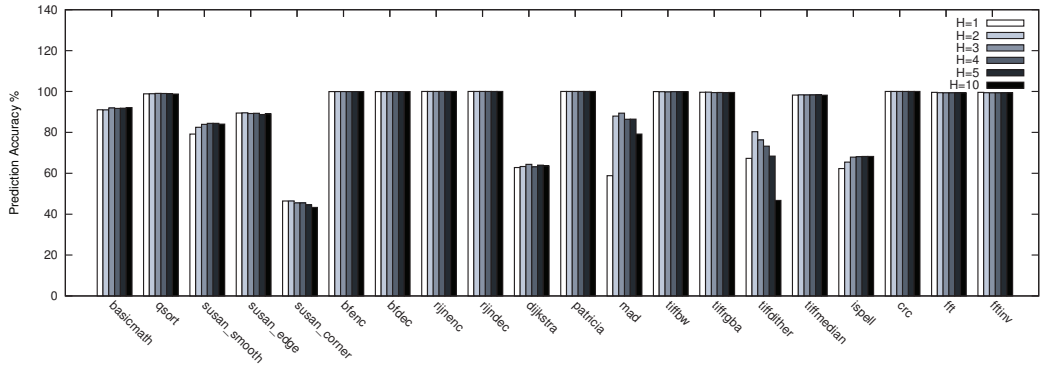


Figure 3.4: *Prediction Accuracy As The Predictor History Size Increases.*

Our 2-level RLE predictor stores a hashed history of the two most recently seen phases and their repetition counts. It looks up this hashed history in a look-up table containing previous such combinations and the phase that followed them to predict

which phase will occur next. As such the predictor is parametrized by the length of its history and the size of the look-up table. A longer history lets it see farther in the past. But, at the same time, a longer history means that the number of possible combinations to store also increases. Which means, for a given prediction table size, there is an increased risk of histories over-writing other histories - aliasing effect. Thus less frequently occurring patterns may evict more frequently occuring ones and degrade prediction. Figure 3.4 shows the prediction accuracy for the N-Level RLE predictor as we vary N, i.e., the history size of the predictor. As we move through the history values of 1, 2, 3, 4, 5 and 10, we see that while for the most part there is no change in prediction accuracy when we increase the history size, there is a big imporvement in it when we increase its value from 1 to 2 in case of `mad` and `tiffdither`. History sizes greater than 2 rarely improve the prediction accuracy and may, on the contrary, affect it adversely e.g., in `susan_corner`, `mad` and `tiffdither`. This can be explained by the fact that the prediction mechanism uses a fixed-size table of 256 entries. As we increase the number of phases kept in the history, the number of possible history permutations increase as well. Thus when the hashing function hashes these history values into one of the 256 indices, it is trying to cram more information into the same number of slots. Thus some less frequently occurring combinations do evict the more frequently occurring combinations that are useful for predictions.

Previous phase predicting studies either did not take into account the warm-up at all [105] or used constant functional warm-up [81]. We, for reasons of practicability, use the detailed simulation for warm-up. Therefore our requirement differs in the sense that we not only need the phase occurrence information in advance but *much* in advance so that we can start the detailed simulation earlier in order to warm-up the simulator for phase arrival. This causes us to predict not one but multiple phases in future so that if we see our desired phase at the end of this prediction

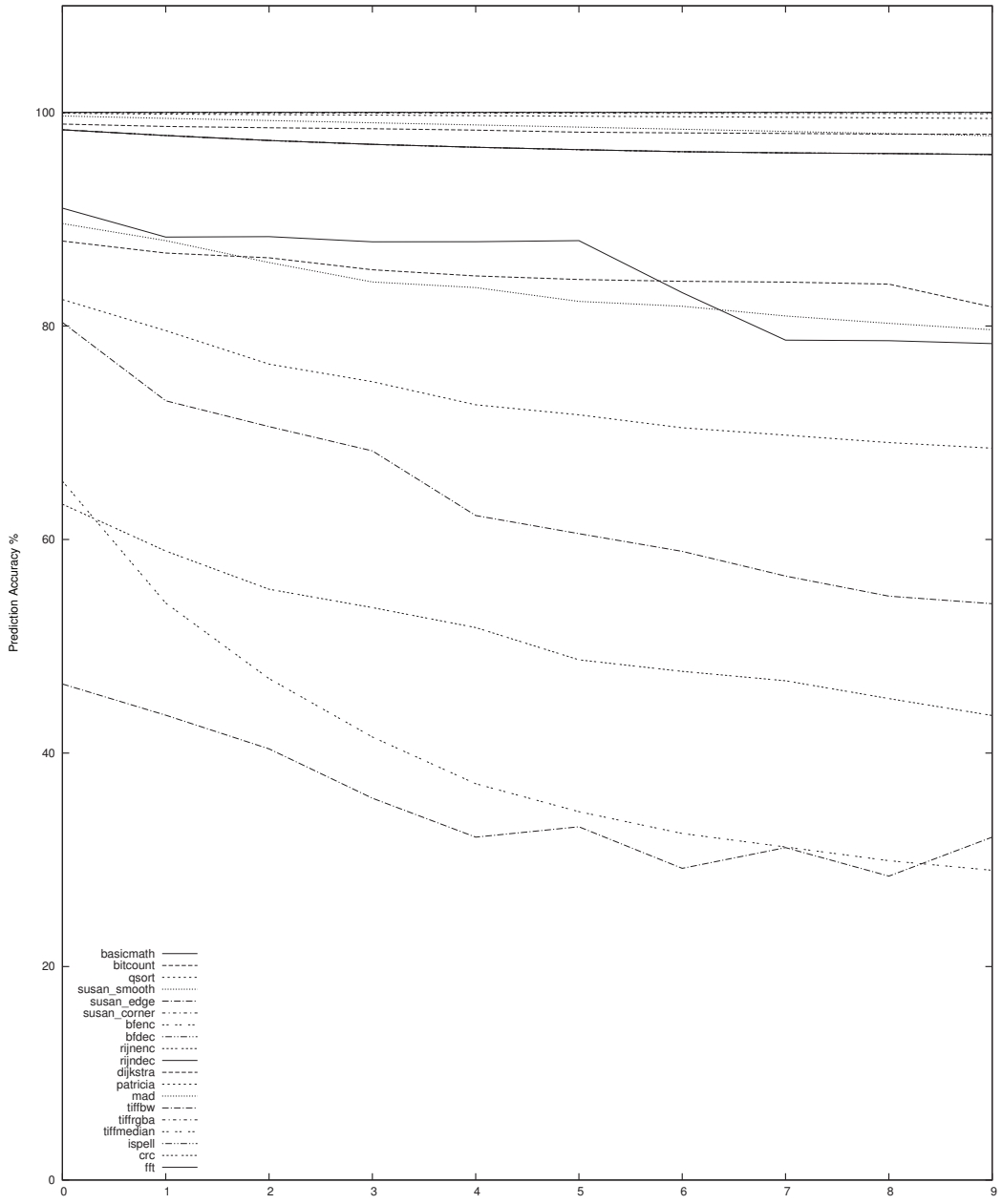


Figure 3.5: *Prediction Accuracy As Predictions Advance Further In Future.*

chain, we could start simulating and use the first few intervals for warm-up. This prediction length should be chosen in such a way that it accommodates the warm-up requirements for all the phases of all the programs under consideration. At the same time, as we try to predict farther into the future, the prediction accuracy decreases

and our predictions are more liable to be wrong.

Figure 3.5 shows what happens when we predict not one but multiple intervals into future based on the same history. As can be seen the history is most accurate for predicting the immediate future and as we venture predicting further, our guesses become less and less accurate. The x-axis shows the index of the interval in the future for which we predict the phase. The y-axis shows the percentage of times that we predicted correctly the phase-id at that index. We see that we predict most accurately for the first index after the history (this is the next interval to come), less accurately for the 2nd one and it gets worse as we proceed up to the 10th interval. For some benchmarks the degradation is gradual while for others it is pretty sharp. The prediction accuracy can be a function of the length of the benchmark, the number of clusters formed and their patterns of occurrence. Hence we have a benchmark with only one cluster, `crc`, and 100% accuracy while there are others which have accuracy closer to 100% and maintain this accuracy even as we predict further into future. On the other hand, there are benchmarks which have low prediction accuracy to start with and it degrades as we venture to guess more deep into the future.

`susan_corner` has the lowest prediction accuracy for the first index, 46.47%, and this drops to 32.12% as we arrive at the 10th interval in the sequence. The low prediction accuracy for the first index is explained by the small size of the benchmark (see Table 3.3), just 421 intervals. The predictor does not have enough time to see different patterns repeat and train itself. The benchmark `ispell` undergoes the worst degradation as we move to predict from the first to the tenth interval, from 65.46% to 29.02%. This is partly due to the complex repetition behavior manifested by its phases.

3.6 Warm-up

We'll be simulating most of our program in functional mode; only launching the detailed mode when we want to simulate an instance of a desired phase. The functional mode is a faster execution mode but it leaves out and does not simulate many architectural details like caches, branch predictors, TLBs, etc. These structures are not updated when simulating in the functional mode. Since the data in these microarchitectural structures have not been updated, they would be different from the ones which would have been there if we had arrived here by way of detailed simulation. This stale data causes unnecessary misses when doing simulation, falsifying the performance measures.

The process of updating the microarchitectural structures with correct data is termed as warm-up. While there exist different propositions for warm-up in literature, those most closely related to our requirements are proposed by SMARTS [110] and on-line SimPoint [81]. SMARTS proposes updating large structures which takes long to warm-up, i.e., caches, branch predictors, etc., during the functional phase of the simulation. When it switches to the detailed simulation mode for performance measurements, it uses the first few thousand instructions to warm-up the smaller structures, i.e., pipeline, etc., and then goes ahead with the measurement. Implementing the functional warm-up poses a few problems, namely:

- Copying all the microarchitectural structures in the functional mode requires significant modifications to the simulator.
- Some structures, e.g., prefetchers, use temporal information for their updates. Functional simulator having no notion of time cannot update them appropriately.
- Functional simulation is fast because it ignores many details of the processor. Adding these structures and their update mechanisms may affect the speed of

the simulator.

The other technique, proposed by on-line SimPoint, is to keep a fixed size trace of memory and branch events at all moments during the simulation and, just before the detailed simulation, to use this trace to update the cache and branch predictor structures. They empirically deduced that a trace of 50000 memory and branch events each is enough for their benchmarks. The problem with fixed warm-up, as it is termed, is that one can under or over estimate the amount of warm-up required. Especially in design space exploration, where one frequently varies the design parameters, it is difficult to find an optimum value that suits every permutation of design parameters. A single value can be too large for a particular architecture causing an unnecessary increase in simulation time or too small for another one incurring a large performance estimation error.

Due to the above mentioned issues in the previous warm-up strategies, we chose to have a warm-up scheme which adapts to architectural changes and to do it during the detailed simulation mode in order to avoid modifications to the functional simulator. SMA [69] is an adaptive warm-up mechanism which uses a bit for each entry in an SRAM structure to determine if it's warm or not. If the number of warm entries in the structures exceeds a certain threshold, it declares the structure warm. Though it has not previously been tested with a sampling approach, we found it to be suitable for our dynamic adaptive warm-up approach.

We use the detailed simulation to do the warm-up. As the detailed simulation advances, the program will access different entries in microarchitectural structures updating them with new data and the simulator state will gradually become warm. The idea is to start the detailed simulation sufficiently in advance of the interval that we want to simulate for performance that by the time we arrive at that interval the micro-architectural structures would be warm. Then we can go ahead and take our measurement. This is the reason we predict multiple future intervals so that we

can "see" our desired phase far enough in future and have time to warm-up.

Each time we launch the detailed simulation, we track the caches with an SMA-style mechanism in order to know how much time does it take to warm-up. And each time we take our performance measurement for a phase, we note down how many intervals it took us to warm-up for it. This value is used to update the *average warm-up length* for this phase in our phase table. Next time we predict a phase sequence, we launch the detailed simulation only if *at least one* of our desired phases is found beyond its average warm-up length. In this manner we know that if we launch the detailed simulation now, by the time we arrive at our desired phase we would have sufficient warm-up. This way in updating the average warm-up length at runtime we make sure that our warm-up strategy adapts itself to each architecture and what's more, to each program phase.

3.7 CPI Calculation

Once we have the performance samples for each part of the program, we would like to average them to get the performance of the whole program. To do this we multiply the average performance for each program phase with a fraction corresponding to its frequency of occurrence. This way we give more weight to the performance of the portions which occur more frequently as they would bear more on the actual performance of the program.

3.8 Principles

The general concept of combining on-line sampling and adaptive warm-up is to select not one, but *multiple continuous intervals* for sampling, and to flexibly allow some to be used for warm-up rather than for sampling purposes. By using a generic implementation of SRAM structures from which a large range of mechanisms can

be derived, the warm-up is shown to require no simulator modification whatever the SRAM-based architecture mechanisms and the sizes of the SRAM structures, and it solely relies on the performance simulator.

Phase identification. The program execution is divided into *intervals* of equal number of instructions. Intervals with similar performance metrics are considered belonging to a *Phase*. SimPoint [92] demonstrated that intervals with similar performance metrics tend to have similar BBV signatures. The functional simulator partitions the simulation into fixed-sized intervals, collects all basic block usage information for each interval and creates a basic block vector for it. It groups intervals with similar BBV signatures into clusters hoping to catch the phase behavior. We identify the phases on the fly, in the spirit of on-line SimPoint [81]. In order to reduce the computational tasks of clustering BBVs on-line, we reduce the dimension of BBVs using a hashing technique. Then we compute the Manhattan distance between each BBV and all clusters identified so far, and decide whether we create a new cluster if the distance between the new BBV and the existing clusters is larger than a threshold distance D, or we consider the BBV to belong to an existing cluster. We adjust the weights of the clusters (numbers of intervals in each cluster) accordingly. Each cluster corresponds to a phase. The whole process is summarized in Figure 3.6.

Predicting multiple phases. After each interval, and based on the information collected so far, the on-line sampling technique must determine when to sample next, i.e., when to switch from functional to performance simulation again. Ideally, at the end of the simulation, for each cluster, there is at least one interval/sample which was performance simulated, allowing to extrapolate these performance metrics to the whole cluster (i.e., the cluster performance is weighed by the cluster weight for computing the overall performance). Therefore, the sampling strategy should permanently monitor the set of all known clusters, and for the clusters without any

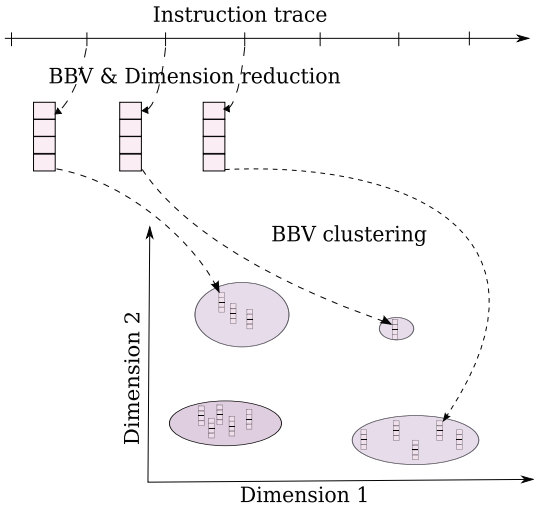


Figure 3.6: *On-Line BBV clustering.*

performance simulated sample yet, it should *predict* its next occurrence (the number of instructions until an interval of that cluster occurs) and trigger performance simulation then, see Figure 3.7. It is important to understand that the cluster id of an interval is known *after* it has been executed, i.e., after the fact, therefore it is indeed necessary to *predict* the occurrence of intervals of a given cluster. The sampling strategy further privileges the clusters with the highest weight so far, i.e., the clusters with the highest number of intervals.

After each interval has been functionally simulated, instead of simply predicting the cluster corresponding to the next interval [81], we predict the *sequence* of clusters corresponding to the N next intervals.

The input of the predictor is a history of the H pairs (cluster id, # of consecutive occurrences) of the past intervals. The output of the predictor are the cluster ids of the N next intervals, see Figure 3.7.

Triggering performance simulation. We then compare this sequence to the list of clusters yet to be simulated. If we find that this sequence will deliver useful additional information, we trigger the performance simulation of that sequence.

The following conditions must all be met for triggering performance simulation:

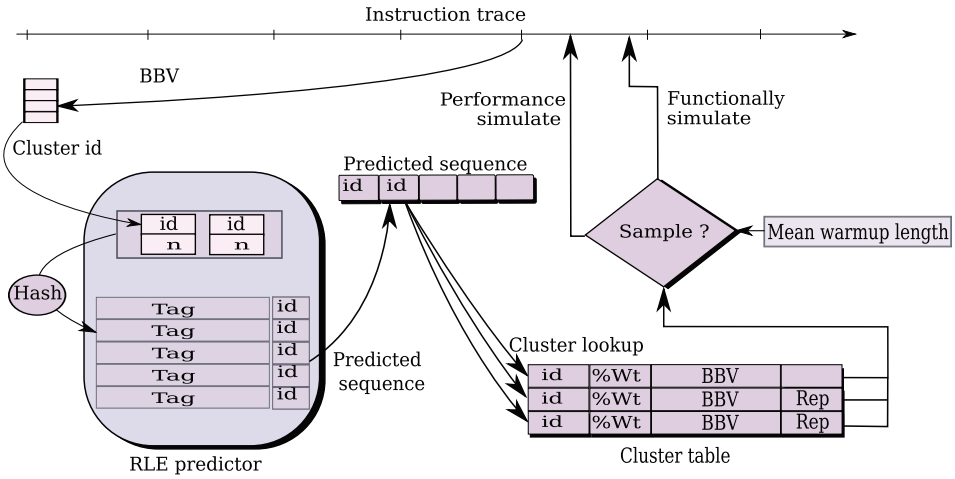


Figure 3.7: *Sampling strategy.*

- The predicted sequence contains at least one cluster with no performance simulated interval yet, and which corresponds to a significant fraction of the program, i.e., at least a fraction $F\%$ of the instructions so far.

- At least one of these "not yet performance simulated clusters" lies beyond the estimated mean warm-up length, i.e., they are located far enough within the sequence not to be part of the warm-up phase.

Warming up. At the beginning of a valid sequence, the performance simulation is activated and the warm-up begins. The only simulator modification that we assume is that all SRAM structures are implemented with our own library component instead of the simple, usual, array declarations, see Figure 3.8. This library component adds one bit per SRAM element (a byte for memory, or any other length for a cache tag array for instance). These bits are reset upon starting a detailed simulation sequence, so that all SRAM elements appear cold in the beginning, whatever the architecture mechanisms they belong to (cache, branch predictor, write buffer, prefetch buffer, etc.). Each time an SRAM element is accessed, the cold bit is reset and a count of the total number of cold vs. warm SRAM elements is maintained. All these operations are transparent to the user thanks to the SRAM library, and

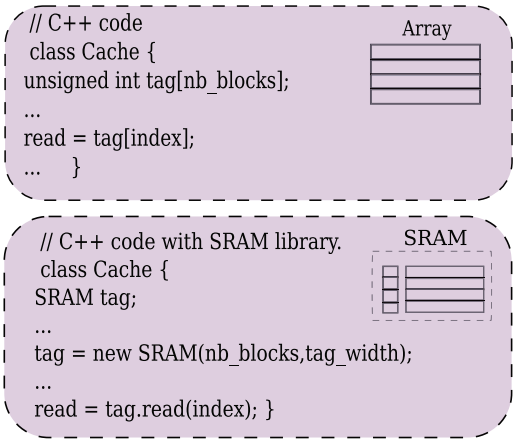


Figure 3.8: *SRAM library.*

such a library is a natural extension to simulation frameworks (SystemC, ASIM, Simics,...). We used our SRAM library with the SystemC simulation framework.

For each interval, the fraction of warmed access to the SRAM structure(s) is monitored, and when this fraction reaches a threshold W, the interval is deemed warm, and the performance statistics can be used, in the spirit of SMA [69]. Unlike SMA, within a sequence of intervals, after an interval has been deemed warm, we keep monitoring the interval warm up status, and a later interval can be deemed cold (and thus its performance statistics ignored). This process is safer than simply asserting that SRAM structures are warmed up after an interval has been found to be warmed.

The warm up length for each sequence is used to compute the mean warm up length, an information used to predict which intervals in future sequences will be used for warm up and which intervals will be used for collecting performance statistics, as explained above. After the warm-up is completed, the performance statistics for all subsequent intervals in the sequence are collected.

Note that it may happen that statistics are collected for clusters that were not sought for, for instance if the warm up length was shorter than predicted. In that case, the corresponding clusters will simply have multiple representative intervals,

which usually improves accuracy (we use the average of measures).

Sampling and warm-up combined. The warm up is adaptive in the sense that if larger SRAM structures are used, it will take more intervals to reach the warm up threshold W. At the same time, the sampling technique can flexibly start at any sample within the sequence. If it turns out that the target clusters have not been simulated, they will remain scored as "high priority" by the prediction strategy and a new sequence containing them will be sought for. Finally, if the mean warm-up interval turns out to exceed N, the simulation sequence is aborted and the intervals simulated are wasted. (Not having an adaptive sequence length is a limitation that we are working on. Ideally, N should be incremented so that the number of warm-up intervals remains smaller than or equal to $N - 1$, i.e., there is at least one interval for which performance statistics can be collected in each sequence.) We discuss this at the end of the section 3.11. **Parameterization.** The strategy relies on a number of parameters that must be set: W the warm up threshold, N the predicted interval sequence length, F the percentage of the total program instructions below which a cluster is considered not important, H the history size used for the prediction, and D the BBV clustering threshold.

We empirically found the overall strategy to be fairly stable for the following parameters. We found that a history $H = 2$ was sufficient across all benchmarks. Two related parameters are the interval length and the minimum sequence length. Even though there is an obvious trade-off for the interval length (too small and the overhead of warming up becomes excessive, too large and not enough intervals can be collected without exceedingly increasing the number of performance simulated instructions), we found our strategy to perform well for intervals ranging from 10000 instructions to 1 million instructions, with 100000 realizing the best error/simulation time trade-off. For that interval size, we empirically found a minimum sequence length of $N = 10$ to be the best compromise.

Two other related parameters are the cluster distance threshold D (recall that a BBV is deemed belonging to a cluster if it differs by less than $D\%$ from that cluster), and F, the percentage of overall program instructions (known so far) that corresponds to a cluster. The smaller D, the higher the number of clusters; this can then be mitigated with parameter F, so that we collect samples only for sufficiently large clusters (accounting for more than $F\%$ of instructions). We empirically found that a good trade-off across benchmarks is $D = 25\%$, $F = 1\%$, in line with other studies [81]. However, because this pair of parameters may be sensitive to the program characteristics, we contemplate using alternative adaptive clustering strategies, such as IDDCA [36], which dynamically adjusts clustering to the target program characteristics.

In the end, the only parameter exposed to the user is W, the warm-up threshold for an interval. This parameter encapsulates the accuracy/simulation time trade-off which is at the core of sampling: the higher W, the higher the number of simulated instructions and the higher the accuracy. We feel the user should be empowered with setting that parameter, even though we provide a default parameter value of $W = 0.1\%$.

3.9 Methodology

Processor	PPC405, in-order
Pipeline	5 stages
DCache	8KB, 2-way 32-byte block
ICache	8KB, 2-way 32-byte block
Registers	32 integer registers
Functional Units	1 ALU, 1 LD/ST
Branch Miss Penalty	2 cycles

Table 3.1: *PowerPC 405 Simulator configuration.*

Our target processor was the PPC405, a 5-stage in-order embedded processor,

see Table 4.1. The processor was implemented using the UNISIM simulation framework [4]. A modified version of SimPoint [92]'s BBV Tracker tool is used to profile the instruction stream and characterize the basic blocks. 5 randomly chosen bits from the start address of each basic block are used as an index into a Basic Block Vector of 32 dimensions. These reduced BBVs are then clustered based on a Manhattan distance threshold (25%). A Run Length Encoding (RLE) predictor [93] is used to hash [46] the last two distinct phases and their number of occurrences in order to index a 256-entry hash table and predict the next phase ID.

We use the embedded benchmark suite MiBench [38] with *large* input sets. In order to demonstrate the resilience of our technique to architectural changes, especially SRAM structures sizes, we vary the cache sizes from 4KB to 64KB. Detailed simulations of the full benchmark suite (no sampling) are used to obtain the actual performance measures, Cycles per Instruction (CPI), of the programs.

After validating the technique on the PPC405 and Mibench combination, we decided to test its resilience in the face of a different architecture/benchmark combination. This time we used the SPEC2000 benchmark suite running on the SimpleScalar simulator. The baseline architecture is detailed in Table 3.2. We test a smaller cache of 8KB and a larger one of 128KB as well. In the next section we detail the experimental results for MiBench. The section after that will list our findings for the SPEC2K suite.

3.10 Experimental Results (MiBench)

In this section, we evaluate the combined on-line sampling and adaptive warm-up technique described in Section 3.8. The two main metrics are accuracy and simulation time. Accuracy is defined as the CPI error of the sampled simulation versus the full simulation. Simulation time is correlated to, and thus defined as, the fraction of the total instructions in the program trace that were performance

Processor	SimpleScalar, out-of-order, 4 way
RUU/LSQ Size	16/8
Pipeline	5 stages
D-Cache/I-Cache	64KB, 2-way 32-byte block
Cache/Memory Latency	2/60 cycles
Memory Ports	2
Registers	32 Int, 32 FP
Functional Units	4 I-ALU, 1 I-Mult 4 F-ALU, 1 F-Mult 4 F-ALU, 1 F-Mult
I-TLB	16 4KB 4-way assoc blocks:lru
D-TLB	32 4KB 4-way assoc blocks:lru
Branch miss penalty	3 cycles miss lat

Table 3.2: *SimpleScalar Simulator configuration.*

simulated (as opposed to only functionally simulated). Since one of the key purposes of our technique is to accommodate architecture modifications, especially SRAM structures sizes modifications, all results are provided for three different cache sizes: 8KB is the baseline PPC405 cache size, and we also experiment with a smaller cache size (4KB) and a significantly larger one (64KB), and compare all results. We also study in more detail the warm-up length, clustering characteristics and prediction accuracy of our sampling+warm-up technique.

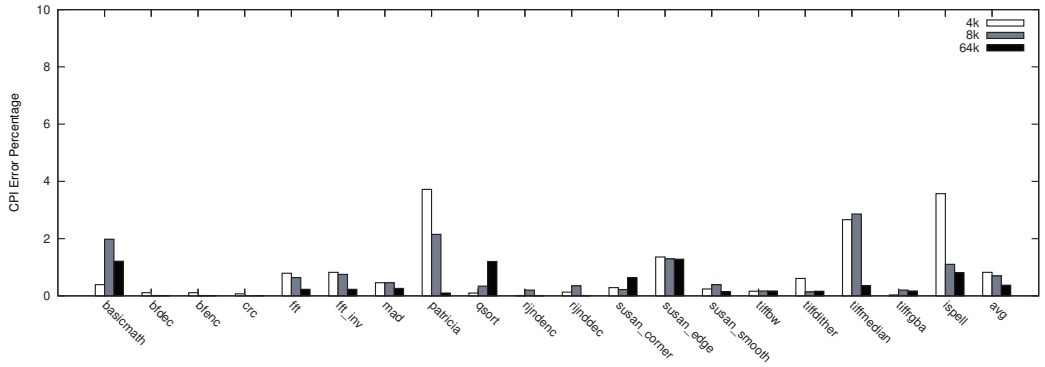


Figure 3.9: *Percentage CPI Error (100k-instruction interval, N=10(25 for* `ispell`*)).*

A note on `ispell`. While all the benchmarks were simulated with N=10, `ispell` shown in these results was simulated with N=25 for the 64KB cache. The reason is since we abandon our simulation sequence without taking a sample as we reach N,

for the 64KB cache, N=10 is never warm enough to take the sample. Therefore the simulation terminates with no samples. N=25 leaves sufficient room for warm-up to take a few samples.

Accuracy. The CPI error is indicated in Figure 3.9 for all benchmarks and the three aforementioned cache sizes. We first note the low average error, less than 0.82%, which is on par with the best sampling accuracy results for non-adaptive techniques [92] or on-line sampling techniques [81]. Moreover, this accuracy is stable across all three cache sizes, and does not increase with the cache size: the accuracy is 0.7% on average for 8KB and 0.37% on average for 64KB. In other words, the accuracy remains stable as an SRAM structure size increases, thanks to the combined dynamic warm-up strategy. To our knowledge, this is the first demonstration of a sampling strategy that exhibits stable accuracy as architecture characteristics change.

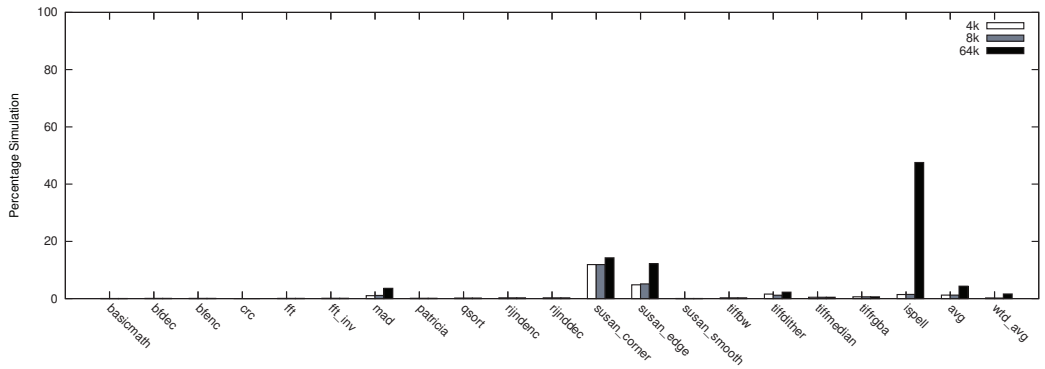


Figure 3.10: *Percentage performance simulated instruction.*

Simulation time. The percentage of simulated instructions is shown in Figure 3.10. Most programs require to performance simulate less than 1% of all their instructions. The fraction of instructions increases with the cache size, due to the larger warm-up required by the larger SRAM structures, but the increase is very moderate. There are a few mentionable points: `ispell` for 64KB cache which simulates 47% in detail (even with N=25, this is explained earlier). Looking in detail shows that

even N=25 is not enough for warm-up in this case most of the time. 42% of the simulation was wasted due to lack of warm-up and only 3% was used for warm-up and samples. `susan_corner` and `susan_edge` also exhibit relatively high performance simulation ratio, between 14% and 12%. However, these two programs are the smallest of all benchmarks (see the total number of intervals in Table 3.3), so that even a few intervals account for a large fraction of the overall simulation; it is interesting to note that the same type of algorithm (`susan_smooth`; `susan` is an image recognition package with several image processing algorithms), with a larger instruction count, has low error and low performance simulation ratio. This is illustrated by the `wtd_avg`, which is the average of the percentage of simulation time weighted by the total number of dynamic instructions in the program: 0.22% for the 4KB and 8KB caches, and 1.6% for the 64KB cache.

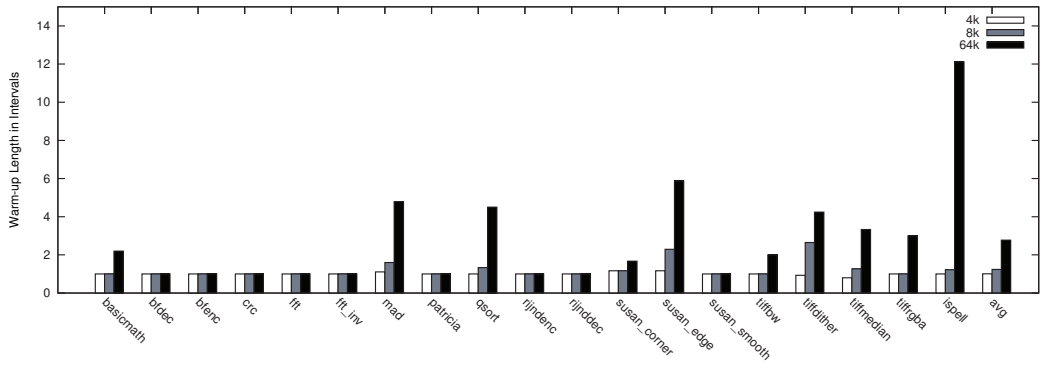


Figure 3.11: *Average number of warm-up intervals per sample.*

Warm-up length. The warm-up length (in number of intervals) is analyzed in more detail in Figure 3.11. For the smallest cache size, all programs require only 1 warm-up interval. For the 8K cache, some programs require a warm-up interval of up to 2 on average. For the largest cache, while most programs accommodate a constant warm-up length of 1 interval, several programs (`mad`, `qsort`, `susan_edge`, `ispell`, etc.) require more than 4 warm-up intervals on average. Especially `ispell`, given N=25, shows an average warm-up length of 12 for the samples it manages to

Program	# Intervals	# Clusters	% Hit
basicmath	83030	8	91.04%
bfdec	20268	3	99.98%
bfenc	20300	3	99.98%
crc	68923	1	100%
fft	75196	9	99.44%
fft_inv	74185	12	99.43%
mad	9562	17	87.95%
patricia	10026	1	100%
qsort	16104	10	98.91%
rijndenc	3509	1	100%
rijnddec	3399	1	100%
susan_corner	421	12	47.51%
susan_edge	1327	18	88.92%
susan_smooth	77692	10	82.48%
tiffbw	4119	4	99.88%
tiffdither	19982	63	80.28%
tiffmedian	14351	81	98.31%
tiffrgba	3460	6	99.65%
ispell	14550	19	65.42%
avg	**27389**	**14.68**	**91.54%**

Table 3.3: *Clustering and clusters prediction.*

take. Therefore, it is not appropriate to use a constant warm-up interval, across architecture configurations, or even across benchmarks for the same architecture configuration.

Clustering and clusters prediction. Table 3.3 shows the number of clusters found for each program and how accurately they were predicted. One can observe that the prediction accuracy is usually inversely proportional to the number of clusters. Intuitively, the larger the number of clusters in a program, the less frequently they will recur during the execution, and thus the lower the probability to find and predict them again. However, the prediction still exhibits an accuracy of 91.54% for 14.68 clusters on average.

3.11 Experimental Results (SPEC2K)

Encouraged by the small error in measuring performance and the low amount of detailed simulation (`ispell` notwithstanding) required to achieve this, we decided to see how the technique fared on other architectures and benchmarks. So we implemented the same mechanism in the SimpleScalar simulator and used the SPEC2000 benchmarks to test it. Using three different cache sizes of 8KB, 64KB, and 128KB, the results are describe below:

N=10. Figures 3.12, 3.13, 3.14 show the perforamnce error, percentage of detailed simulation, and the average warm-up lentgh when we apply the adaptive representative sampling to the SPEC2K benchmarks with a predicted sequence length of 10.

Now SPEC2K benchmarks are quite different from MiBench ones in terms of their size, memory footprint and phase behavior. This can be seen from Figure 3.12 which shows an average error of >5% for all cache sizes. The error for many benchmarks goes up to 10% with `vpr` showing the highest error of 26% for the 128KB cache size.

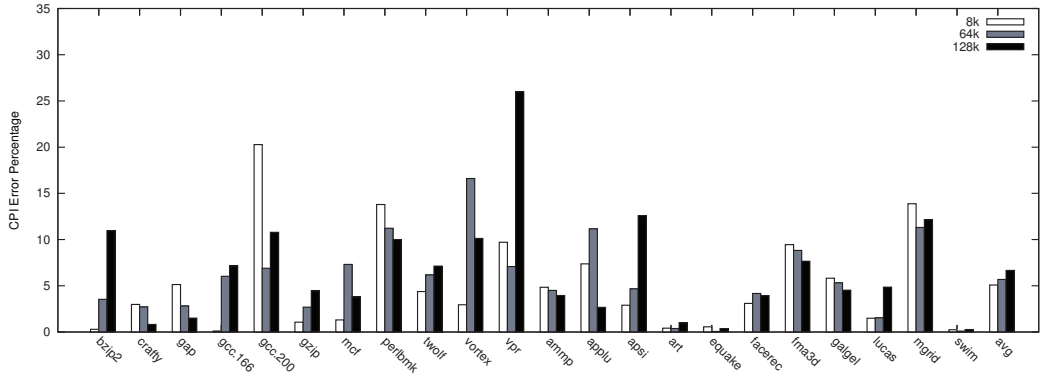


Figure 3.12: *Percentage CPI Error (100k-instruction interval, N=10).*

A look at the percentage detailed simulation in Figure 3.13 reveals that while the portion of program simulated in detail remains low for 8KB and 64KB caches, for

the 128KB one the average is 2.5% with three benchmarks crossing the 10% mark. This excessive amount of detailed simulation can be traced to the inflexible sequence length of N=10. Most of the detailed simulation is wasted because at the end of most sequences we abandon the detailed simulation mode, without taking a sample, because of a lack of warm-up. Table 3.4 shows the amount of detailed simulation and the wasted part of it as percentages of the whole program. We can see that whereever a benchmark shows large amount of detailed simulation, a major part of it consists of intervals wasted due to sequence termination before we could achieve our desired warm-up threshold.

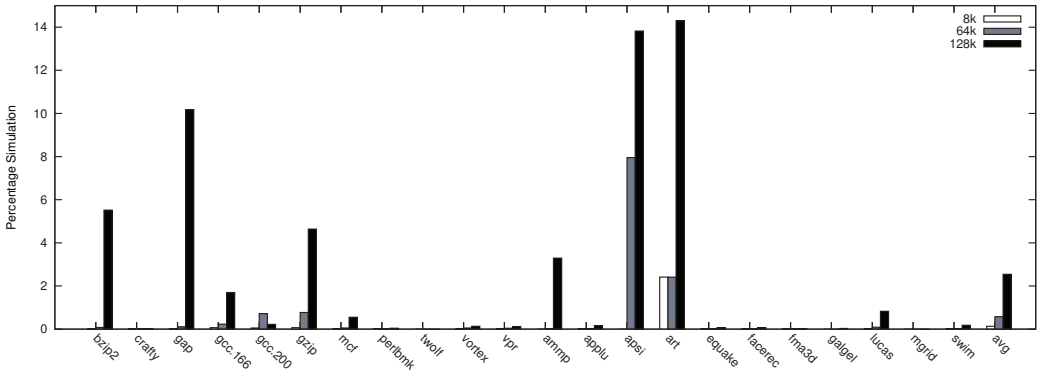


Figure 3.13: *Percentage performance simulated instruction (100k-instruction interval, N=10).*

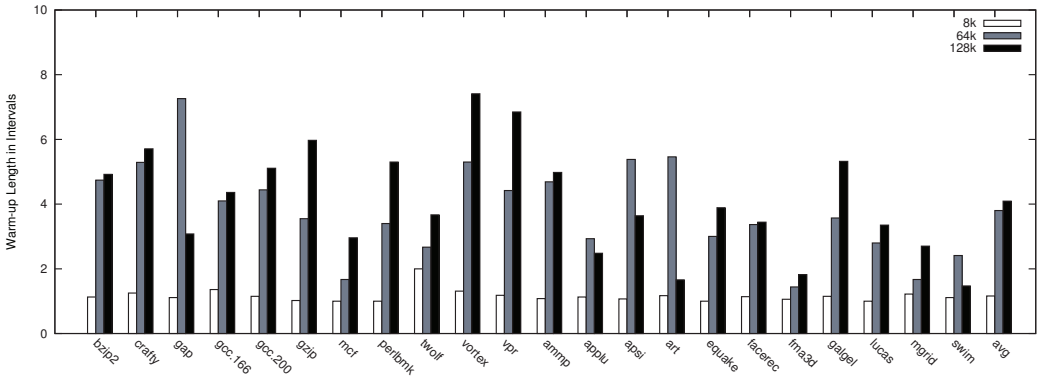


Figure 3.14: *Average number of warm-up intervals per sample (100k-instruction interval, N=10).*

Program	8KB simulated	8KB wasted	64KB simulated	64KB wasted	128KB simulated	128KB wasted
bzip2	0.02	0	0.08	0.04	5.52	5.13
crafty	0.01	0	0.01	0.01	0.01	0.01
gap	0.01	0	0.1	0.09	10.18	10.12
gcc.166	0.07	0	0.23	0.09	1.7	1.45
gcc.200	0.05	0	0.72	0.61	0.22	0.15
gzip	0.07	0	0.77	0.03	4.64	2.41
mcf	0.02	0	0.05	0.03	0.55	0.12
perlbmk	0.02	0	0.02	0	0.04	0.02
twolf	0	0	0	0	0	0
vortex	0.01	0	0.05	0.02	0.13	0.11
vpr	0.02	0	0.04	0.01	0.12	0.11
ammp	0	0	0.01	0.01	3.29	2.92
applu	0.01	0	0.01	0	0.16	0.08
apsi	0	0	7.95	7.66	13.82	13.43
art	2.41	0	2.41	0	14.3	0.89
equake	0.01	0	0.01	0	0.07	0.06
facerec	0.01	0	0.02	0.01	0.06	0.05
fma3d	0.01	0	0.01	0	0.02	0.01
galgel	0	0	0	0	0.03	0.03
lucas	0.01	0	0.09	0.08	0.83	0.75
mgrid	0	0	0	0	0	0
swim	0.01	0	0.01	0	0.18	0.16
avg	0.13	0	0.57	0.39	2.54	1.73

Table 3.4: *Portion of simulation wasted.*

Figure 3.14 shows the average warm-up lengths for the SPEC2K programs. We see that in general the warm-up lengths are larger than those seen for MiBench programs. This is because of the larger cache sizes and the larger memory footprint of the SPEC2K programs. It can also be remarked that the average number of intervals required to achieve warm-up increases with the increase in cache size. This was expected as larger caches take longer to warm up.

N=25. Since we noted that N=10 was not enough for some programs when simulating a cache size of 128KB and we had to abandon the detailed simulation sequence before we could achieve our desired warm-up, we tried to manually increase the sequence length to N=25 to see the effect on warm-up and the resulting detailed simulation percentages. The data is shown in Figures 3.15, 3.16, 3.17 representing the perforamnce error, detailed simulation percentage and average warm-up length

respectively.

The performance error has not changed much. This reflects the inherent difficulty in capturing the phase behavior of the SPEC2K programs which are more complex than the MiBench ones. One noticeable fact is the increase in the error for `galgel`. It now shows an error of 34% compared to the previous one of 4.5% for N=10.

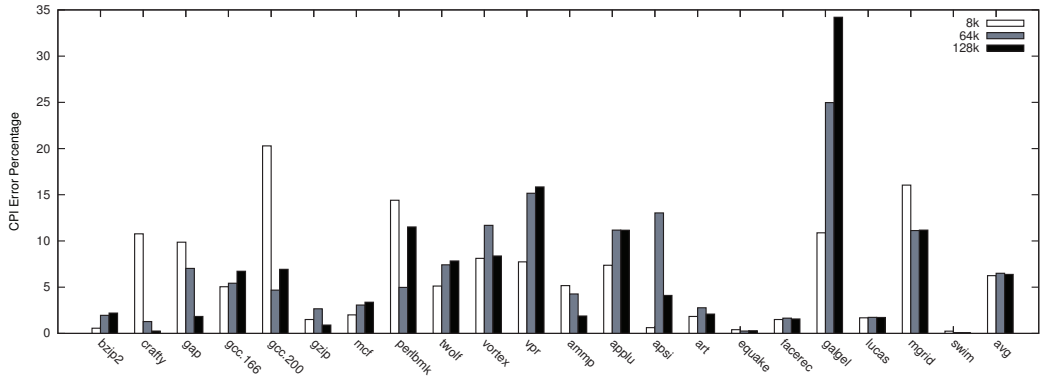


Figure 3.15: *Percentage CPI Error (100k-instruction interval, N=25).*

First of all we see from Figure 3.16 that the percentage of detailed simulation is greatly reduced for most benchmarks for the 128KB cache. This is due to the fact that increasing the sequence length from 10 to 25 lets give more room to the simulator for warm-up. This results in less number of sequences being abandoned and the detailed simulation that was previously wasted is now part of the warm-up and serves to provide a sample. `apsi` still shows a detailed simulation percentage of 9.5% but that is because even a sequence length of 25 is not enough for its warm-up. We see that the wasted simulation percentage is still 8% of the program. This highlights the difficulty of manually setting the sequence length and underscores the need for an adaptive mechanism.

Table 3.5 shows the total number of intervals, the number of clusters formed, and the prediction accuracy for each of the SPEC benchmark tested. Note the difference between the prediction accuracy for the SPECINT (the first 11) and the SPECFP benchmarks. The SPECINT benchmarks exhibit a relatively complex

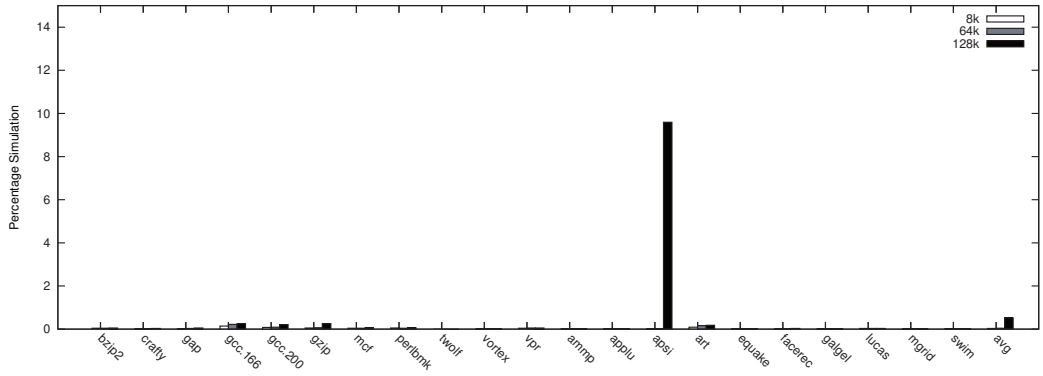


Figure 3.16: *Percentage performance simulated instruction (100k-instruction interval, N=25).*

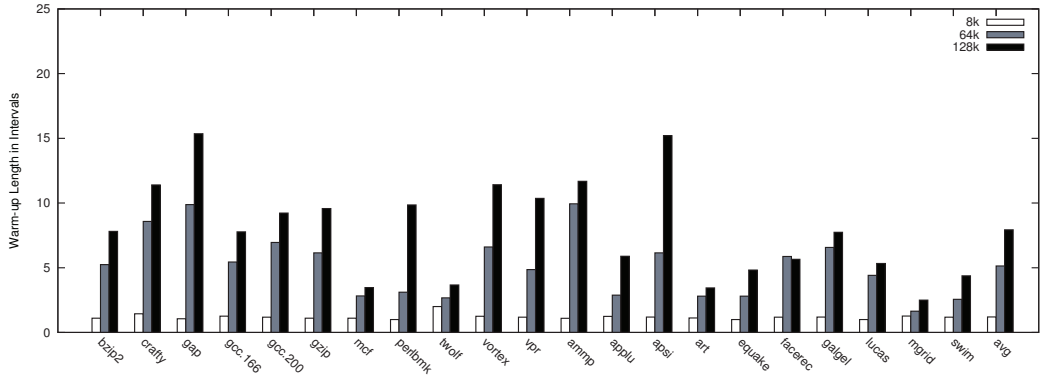


Figure 3.17: *Average number of warm-up intervals per sample (100k-instruction interval, N=25).*

phase behaviour - and thus have lower phase prediction accuracy - compared to the SPECFP ones which are fairly regular.

Fixed vs adaptive sequence length. As we saw for `ispell` in MiBench and for most of the SPEC benchmarks, fixed sequence length of N=10 was inadequate for large cache sizes. The reason is that the length of a sequence limits the number of intervals used for warm-up. As the caches grow, it takes more time to warm them up. Thus it may happen that a cache may never get warm because of a small sequence length. Secondly choosing too large a sequence length will result in doing unnecessary detailed simulation and defeat the purpose of simulation acceleration. It is also difficult to manually adjust sequence length to each architecture/program

Program	# Intervals	# Clusters	% Hit
bzip2	1.09E+006	108	67.41%
crafty	1.92E+006	29	51.12%
gap	2.69E+006	68	93.96%
gcc.166	469177	207	71.75%
gcc.200	1.09E+006	364	61.3%
gzip	843673	94	68.87%
mcf	618674	47	89.73%
perlbmk	399392	37	68.33%
twolf	3.46E+006	96	99.9%
vortex	1.19E+006	21	64.68%
vpr	840687	82	42.32%
ammp	3.27E+006	84	65.71%
applu	2.24E+006	44	99.02%
apsi	3.48E+006	57	98.82%
art	417951	54	71.48%
equake	1.32E+006	36	98.75%
facerec	2.11E+006	42	76.63%
fma3d	2.68E+006	82	98.91%
galgel	4.09E+006	94	96.76%
lucas	1.42E+006	38	98.71%
mgrid	4.19E+006	41	98.59%
swim	2.26E+006	34	99.32%
avg	**1.91E+006**	**79.95**	**81**

Table 3.5: *Clustering and clusters prediction.*

pair. As we saw that increasing the sequence length from N=10 to N=25 did the trick for most of the SPEC2K benchmarks but still was insufficient for `apsi`. Therefore the need for an adaptive sequence length which adapts the length of the detailed simulation to the warm-up requirements of the program/architecture is evident.

3.12 Conclusion

In this chapter we demonstrated that representative sampling can be combined with adaptive warm-up in a sampling technique which is both user agreeable and architecture agnostic.

We demonstrated that an on-line phase detection fares well when classifying the program behaviour most of the time and that reducing the BBV dimensions to a certain limit can capture the program phase behaviour while reducing the computing

cost at the same time. Secondly, we showed that we can sacrifice the accuracy of the off-line clustering mechanisms in the favor of simple on-line mechanisms with a tolerable hit on the performance accuracy while gaining on the computation side.

We also extend the phase prediction mechanism to predict over multiple future intervals. In doing so, we give our technique room for warm-up before it samples its desired interval. Though it was observed that for some benchmarks, with complex phase behavior, prediction accuracy decreases as we try to predict far in future.

We find detailed simulation a better alternative for warm-up than other methods because it avoids modifying the functional simulator and also our warm-up technique adapts to architectural needs. A related important issue is the length of prediction sequence. If the predicted sequence's length is larger than the maximum average warm-up length possible, then the warm-up will easily adapt to the program/architecture needs. But in practice we found that it was difficult to select a sequence length such that it satisfies the warm-up requirements and, at the same time, keeps the detailed simulation to a minimum. An alternative is to make the predicted sequence length variable such that it adapts to the simulator needs at run time. This is the next logical extension of this work and a focus of our ongoing efforts.

Over all we present a technique which makes it possible to do a single-run sampling performance estimation, taking into account the warm-up adaptability to architecture changes, significantly reducing the simulation times. Our technique achieves an average CPI error of less than 0.82% and requires a detailed/performance simulation of less than 1.6% of the program instructions on average, for the benchmarks and architecture configurations considered. Moreover, because our technique does not require any modification of the functional simulator, except for the generic collection of basic block vectors, it is entirely compatible with fast functional simulation techniques such as binary translation.

While the mechanism presented in this chapter works for the most part, as

shown in Section 3.11, the current implementation is not always adaptable to drastic changes in architecture. The main reason is the inflexibility of prediction sequence length and the degradation of prediction accuracy as the said sequence length and the number of clusters increase. This prompted us to explore the other alternative to representative sampling i.e., statistical random sampling, which we present in the next chapter.

Chapter 4

Transparent Statistical Sampling

4.1 Introduction

In the previous chapter we discussed Representative Sampling as a mean to select the portions of the program we'd like to simulate. In this chapter we introduce Statistical Sampling as an alternative method of sample selection.

Sampling, as discussed earlier, is a common technique which uses a subset of observations within the population to predict the properties of the whole population. Using a subset of the population is useful due to its lower cost and faster data collection. Sampling theory has proved that even a small portion of the population, if properly selected, can give a fairly accurate idea of the population properties[66].

In our experiments we divide the instruction stream of the program into intervals of fixed length (in number of instructions). This sequence of intervals, or more exactly their *cycles per instruction (CPI) values*, make up our population. Keeping in line with our target of performance simulating a minimum number of instructions, we'd like to know the properties (CPI) of this population of intervals without having

to performance simulate the whole instruction/interval stream. Statistical sampling theory seems to offer insights as to which intervals to simulate in detail in order to obtain a performance (CPI) estimate of the whole population (program).

A relevant question that needs attention, after the selection of *intervals-to-simulate*, is the state of the simulator before our sample. In order to accelerate the simulation process most of the time we would use the functional simulator to arrive at the desired point in execution. The functional simulator, in order to go fast, ignores and does not update the microarchitecture structures. As a result the data in those structures is leftover from the previous detailed simulation. This stale/false data will bias our performance measurement and affect the results. We need to warm-up these structures (fill them up with correct data) if we want our measurements to be reliable.

Again, when implementing statistical sampling, we have taken care to devise a mechanism which needs the user to intervene in a minimal fashion. Also that the system should adjust smoothly to changes in hardware and software characteristics. Since it centers around our objective of an adaptive and usable technique, we call this approach *Transparent Random Sampling* and the implementation *Transparent Sampling Engine (TSE)*.

All these issues (interval selection, warm-up, and usability) are addressed in the subsequent sections.

4.2 Interval Selection

As opposed to representative sampling, random statistical sampling advocates the use of an unbiased or random subset of observations/measurements within the population. The properties of this sample set can then be extrapolated to estimate the population characteristics. The field of statistics has well established procedures to ensure the correctness of these measures. A well chosen sample of appropriate size

reflects the population properties as a whole. The goal of Sampling is to select such a representative but minimal sample.

If we consider our program execution as a series of instructions being executed one after the other and imagine this instruction stream as our population, it becomes clear how statistical sampling is applicable to our case. It would suffice to know the CPI of a few randomly selected instructions in detailed mode to calculate the CPI of the whole program. Only, the cost of switching to detailed mode is too big to incur it to simulate only one instruction. So instead of considering our execution stream being composed of individual instructions, we consider it being made up of *groups of instructions* called *intervals*. The size of all intervals is same in terms of number of instructions. Our scheme now boils down to randomly selecting and simulating in detail a minimum number of intervals from our program execution stream so that we can get an accurate estimate of the performance of the program.

A recapitulation of some basic Statistics will help clarify this process.

4.2.1 Statistical Distributions

Statistical calculations are based on certain assumptions about the distribution of data in the population. Populations are classified into *distributions* based on the probability of finding data in a certain position. This *probability distribution function* characterises the spread of population data and distinguishes one distribution from other.

One of the most commonly occurring distribution of data in nature is the Normal Distribution. Figure 4.1 shows what a normal distribution looks like. It's a bell shaped curve with data spread out evenly on both sides around the center. The x-axis shows the values that the data in the population can take and the y-axis shows their frequency of occurrence normalized to 1. As can be seen from the peak in the curve, values near the center occur most often and as we move away from the

center the frequency of occurrence decreases.

The probability distributions, with the same probability distribution function, can be characterised by their certain properties. A mean μ, also know as the average value, is the sum of observations divided by the number of observations:

$$\mu = \frac{1}{n} \times \sum_{i=1}^{n} x_i,$$

where n is the size of the population. Also known as the expected value, in a normal distribution, values close to this value are the most probable to occur. In Figure 4.1 this is the value in the center on x-axis.

More than one normal distributions can have the same mean. In this case they can be distinguished by their variance. Variance of a distribution is given by the formula:

$$var = \frac{1}{n} \times \sum_{i=1}^{n} (x_i - \mu)^2.$$

It defines the spread of the data around the mean. The higher the variance, the more spread the data is about the mean and the flatter the curve of the distribution and vice versa.

A related quantity is the standard deviation of the distribution. It is defined as the square root of the variance.

$$\sigma = \sqrt{var}.$$

Normal distribution has been well studied and we know that approximately 68% of the values are found in the region within one standard deviation from the mean, 95% within 2 standard deviations and 99.7% of the values are found within 3 standard deviations from the mean. This is demonstrated by the figure.

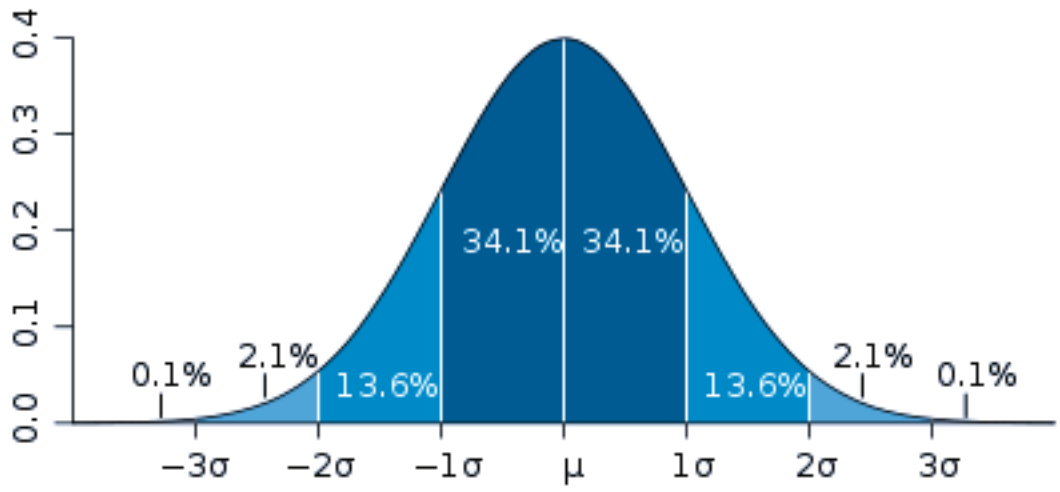


Figure 4.1: *Normal Distribution*

With this information in hand, given a value from a normal distribution, we can say that it has a 95% chance of being within 2 standard deviations from the mean. Of course, we'd be needing the values of the mean and the standard deviation for the population to make this statement meaningful. That's where sampling theory kicks in. Since, under everyday use, it's usually very difficult to obtain these values for the entire population, we use a sample subset of the population and use the mean and standard deviation of this sample as approximates of the mean and standard deviation of the population. The above equations become:

$$\overline{x} = \frac{1}{k} \times \sum_{i=1}^{k} x_i$$

and

$$var = \frac{1}{k} \times \sum_{i=1}^{k} (x_i - \overline{x})^2$$

and

$$s = \sqrt{var},$$

k being the number of elements in the sample.

When characterising population properties using samples, it is difficult to get them 100% right because however good a sample, it always contains less information than the whole population. Therefore instead of giving a single value for that parameter, intervals likely to contain the value of that property are usually used. A confidence interval (CI) is one such interval. It helps estimate the reliability of the measure. After a sampled measurement, the result can be usually announced as: we can say with α% confidence that the value of the parameter lies in the interval $[X, Y]$. A confidence level of 95%, for a confidence interval $[X, Y]$, would mean that if measured a large number of times, the values of the desired property would lie within the interval $[X, Y]$ 95% of the time. Given the standard deviation (s), sample size (k) and the z-value (c) for a distribution, the confidence interval can be estimated by:

$$CI = (\overline{x} - [c \times \tfrac{s}{\sqrt{k}}], \overline{x} + [c \times \tfrac{s}{\sqrt{k}}]).$$

The z-value (c) is the number of standard deviations above or below a mean, we expect to find a given value.

The confidence interval is centered around the mean $\overline{x}$ of the sample and we expect the real mean μ of the population to lie within the boundaries of this interval. If we knew the real mean of the population we could have calculated the error in our estimation by comparing the sample mean to the population mean:

$$Percentage\ Error = \tfrac{abs(\overline{x}-\mu)}{\mu} \times 100.$$

In practice we do not know the real mean CPI of the population comprising all the intervals in the program (we have no need of simulation in that case). What

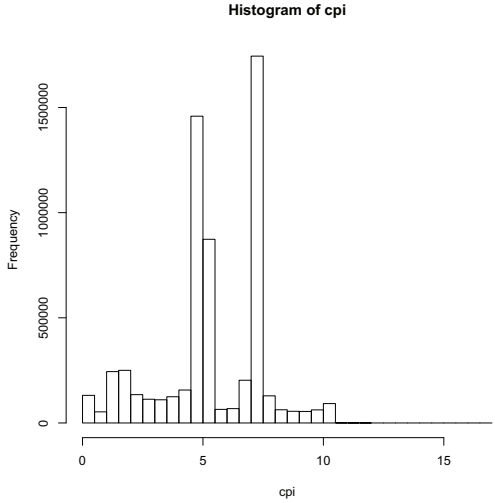
we do know, however, is that once we have calculated our confidence interval $[X, Y]$ around the sample mean $\overline{x}$, the population mean μ should lie somewhere in that interval. Since the maximum difference this *real* mean can have with the sample mean is when it equals either of the boundaries X or Y, we can calculate the theoretical maximum possible error as:

$$\textit{Statistically Estimated Error} = \frac{abs(X-\overline{x})}{\overline{x}} \times 100.$$

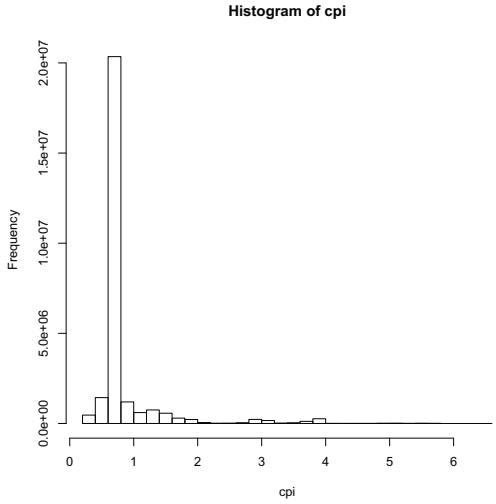
What is important here to note is that since the user has no way of knowing whether his sampled measurement is correct or not, this *Statistically Estimated Error* provides an estimate of the reliability of his result. The larger this value, the larger is the confidence interval and the farther apart the real mean CPI μ can be from our experimentally measured CPI $\overline{x}$. Thus large values of *Statistically Estimated Error* can be an indication of a less reliable result and a potentially large error.

Like other characteristics of a probability distribution, the confidence interval is also a parametric measure and makes certain assumptions about the underlying population. The above equation assumes a normal distribution and that the measurements are done completely randomly independent of each other.

Since many of the statistical parameter calculations make assumptions about the normality of the underlying populations, it's a logical move to verify those assumptions. With the execution stream of our programs divided into intervals of equal length, we simulated the SPEC2000 benchmarks *in detail* and noted the CPI for each interval. Using the R [86] programming language we built histograms for these CPI populations to see their shapes. The Figure 4.2 shows the histograms of the CPI for two of these programs. By contrasting these shapes with the *Normal Distribution* in Figure 4.1 we can observe that the distributions of the CPIs for these programs are hardly normal. The figure for `mcf` exhibits two very sharp peaks side by side while the distribution for `gap` is clearly skewed to the right.



(a) mcf



(b) gap

Figure 4.2: *CPI distributions*

The concerns caused by the programs' CPI distributions not following the normal distribution are allayed by the Central Limit Theorem (CLT) [45]. The Central Limit Theorem states that the mean, or sum, of a sufficiently large number of independent random variables, each with finite mean and variance, is approximately normally distributed irrespective of the original distribution of the population. An

irregularly distributed population undermines the CI's assumptions and prevents us from applying the CI calculations to an individual value as that value will follow the population's distribution instead of the normal one. But we can apply the CI calculations to the mean of a sample subset of the population as, according to the CLT, the mean of sufficiently large sample will follow a normal distribution.

The CLT is a central tenet of the probability theory and helps explain the abundance of the normal distribution in nature. As many natural phenomena are the sum of unobservable random events, these sums are observed following the normal distribution. A point of discussion in CLT has been the sample size. What does "sufficiently large" mean? In literature we can find references which would believe a size ($k > 30$) to be sufficiently large for all purposes. We decided to put this claim to test. The benchmark `gap` exhibits an unusually skewed distribution of CPI. We have plotted the CPI mean distributions for different sample sizes in figure 4.3. We start from a sample with only 5 measurements and go up to 200. For each sample size we take 500 samples from the original CPI distribution and then plot the distribution of the means of these 500 samples in the figure. For small sample sizes we observe that the distribution of sample mean follows the same skewness as observed in the original CPI distribution. Even when the sample size exceeds 30 ($k = 50$), traces of skewness remain. For this particular benchmark we observe that from a sample size of 100 the sample mean's distribution starts resembling the normal distribution. In our experiments the samples always contain more than 100 measurements so we can safely apply the confidence interval calculations assuming the normal distribution of means.

4.3 Warm-up

Having addressed the question of interval selection for detailed simulation, we turn our attention to the second question pertaining to the correctness of the microar-

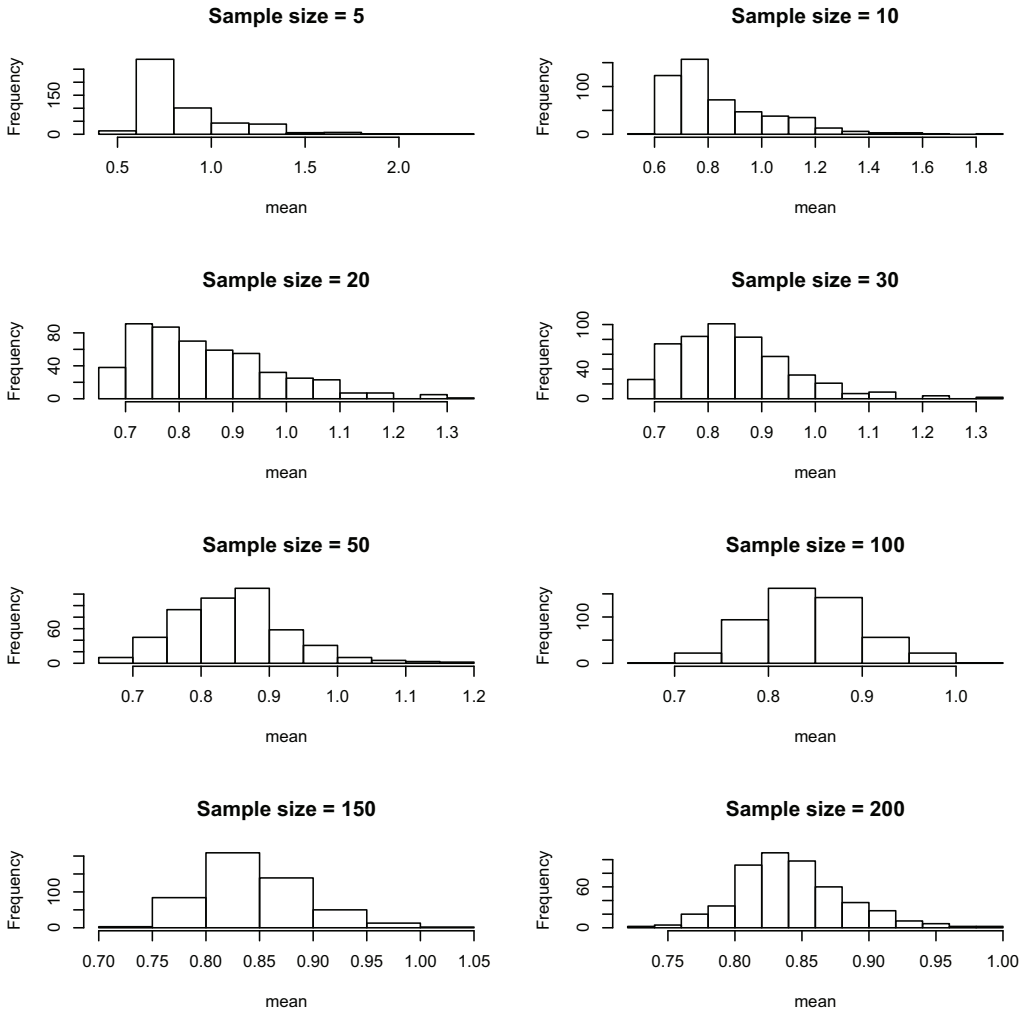


Figure 4.3: *Distribution of mean as a function of sample size for* `gap`.

chitectural state of the simulator before we start the detailed simulation.

In order to accelerate benchmark simulations, Sampling techniques rely on simulating most of the program in the faster functional simulation mode and switching to the slower performance simulation mode only when trying to take performance measurements. During the functional simulation mode, the microarchitectural state of the processor, i.e., caches, branch predictors, TLBs, etc., is not updated. Therefore upon switching to the performance simulation mode after a spell of functional

simulation, the data in these SRAM structures are outdated (they are the remnants of the previous performance simulation mode). Thus the first accesses to these structures would find invalid data and result in misses incurring extra cycles. This would cause an over estimation of the CPI. These structures at the end of a functional simulation, with invalid data, are referred to be in the "cold" state.

Once the performance simulation mode is launched, the accesses to these structures cause misses and result in the progressive update of the respective fields with correct data. When all the fields of a structure are updated, we consider it completely "warm". Sampling softwares try to base their measurements on warm structures as they are free of bias introduced by cold start misses and represent correctly the actual performance of the program.

4.3.1 Implementing the Warm-up

We now consider which warm-up and sampling approaches are compatible together. As noted before, on-line representative sampling, when combined with either static or adaptive warm-up, gave mixed results. For this reason, our approach titled Transparent Statistical Sampling (TSS) is based on statistical sampling, with samples randomly selected.

However, we do not want to resort to continuous warm-up as in SMARTS in order to avoid functional simulator-level modifications; we want to warm up SRAM structures just using the performance simulator. Moreover, for time-sensitive mechanisms, such as prefetching [41](e.g., to warm-up prefetch buffers), it can be difficult to implement the warm-up altogether since the functional simulator has no notion of time.

Fix-sized detailed warm-up methodologies are not transparent to architectural changes. A fixed number of detailed instructions, while they may be perfect to warm a certain size of SRAM structures, will prove insufficient for larger and an overkill

for smaller sizes.

Checkpointing [108][109] and adaptive warm-up techniques [47][28] require the user to execute the program multiple times. This runs contrary to our principle of the ease of usability. Checkpointing also may require large disk space for storage.

Therefore, we turn to SMA[69]. The difficulty is then to design a joint statistical sampling + adaptive warm-up technique which will deliver good performance, i.e., high accuracy and small fraction of total instructions simulated, and still remain entirely transparent to the user, i.e., no parameter to set except ones easily comprehensible by the user.

In order to achieve transparent sampling, we warm up using the performance simulator: by simply running, the performance simulator will progressively warm up the SRAM structures. The simulation is partitioned into fixed-size intervals of N instructions each. At any interval, we assess the degree of warm-up. For that purpose, we monitor the load/store references that access SRAM entries which are already warmed, as in SMA [69]. In order to transparently implement this monitoring we provide a simple SRAM library (in C++ or C) on top of which any cache or table can be easily implemented. This is the only simulator modification required to use Transparent Sampling, and it is benign: the user only needs to replace the classic array or object declaration used for SRAM structures with an SRAM class instantiation (or a call to a function in C), see Figure 3.8. This SRAM class implements a single bit per SRAM entry. This bit is reset by the Transparent Sampling Engine (TSE) before each sample, and set at the first reference within a sample. Thanks to this bit, it is possible to determine whether each access (e.g., load/store, branch prediction table access, etc.) is cold or warm without any further simulator modification by the user.

So far, this warm-up technique is very similar to SMA [69], except that we provide the library and class support. Unlike SMA, we do not use the fraction of

SRAM structures which are warmed as a warm-up criterion, because we empirically observed this criterion to be highly sensitive to the program behavior. For instance, some program parts with a small workload will only warm up a fraction of the cache. The other option was to monitor the memory accesses during an interval and count the fraction of warm accesses. SMA combines both warm-up criteria but we found that solely using the fraction of warm accesses was more robust. We deem that all the SRAM structures are warmed when the fraction of warm accesses to each of these SRAM structures (e.g., the different caches of a memory hierarchy) is above a threshold.

However, even though our adaptive warm-up technique makes use of the aforementioned warm-up criterion, it does not rely on the variable-length warm-up interval of SMA for the following reason. We found that SMA can have a non-trivial impact on the *selection of intervals used for performance measurement*, the interval immediately following the warm-up intervals; more exactly, that it could shift the performance measurement intervals in a way that could be degrading the randomness of performance measurements (a key aspect of statistical sampling). Because SMA was studied as a warm-up technique alone (as opposed to a warm-up + sampling technique), it is only normal that this effect has gone unnoticed, but we found it to be severely detrimental in some cases. Consider a program where the following pattern recurs often: a region with accesses to many distinct addresses followed by one or several region(s) with repeated accesses to a few addresses. The first region is likely to breed significantly more cold accesses than the second region(s). As a result, the fraction of cold accesses in the first region will be high, and the warm-up threshold won't be passed. When the program enters the second region(s), the fraction of warm accesses quickly increases because just a few addresses are being repeatedly used, and the threshold is likely to be passed. As a result, adaptive warm-up has shifted the performance measurement interval to a region with few,

repeated accesses. Now, the first region is likely to exhibit a higher *miss rate* than the second region because a higher number of distinct addresses are accessed. As a result, adaptive warm-up has *shifted the performance measurement interval to a region with a lower miss rate*.

Note that, in a more complex case where a many-address region follows a few-address region which follows again a many-address region, adaptive warm-up could have the exact opposite effect and shift the performance measurement interval to a many-address region, artificially increasing the measured miss rate. We observed both cases. Note that these cases are not frequent, and adaptive warm-up using only the threshold of warm accesses often works well; but in some cases, this bias severely degrades the accuracy of sampling, making the technique less robust.

4.3.2 Average Warm-up Size

So we need to avoid the shifting effect of adaptive warm-up, but at the same time, we do need adaptive warm-up in order to adapt to variable SRAM sizes. In order to reconcile both constraints, we proceed as follows. At each sample we measure the number of intervals it takes to warm-up, necessary to pass the threshold, and using all such measurements since the beginning of the execution, at any sample, we compute the *average warm-up size*. At the next sample, we use this average warm-up size as the warm-up length, independently of the threshold. After a few samples, the average warm-up size stabilizes, and this is almost akin to a fixed-size warm-up. As a result, performance measurement intervals are shifted by an almost constant number of instructions, avoiding to bias the randomness of their selection. Even though we do not factor in the threshold for stopping the warm-up, we monitor it. In case the warm-up threshold has still not been reached after the performance measurement interval, *we let the performance simulation carry on until the threshold is reached*, see Figure 4.4, but we do not use the corresponding

intervals for performance measurement nor warm-up; they are simply used to allow us to compute the new average warm-up size.

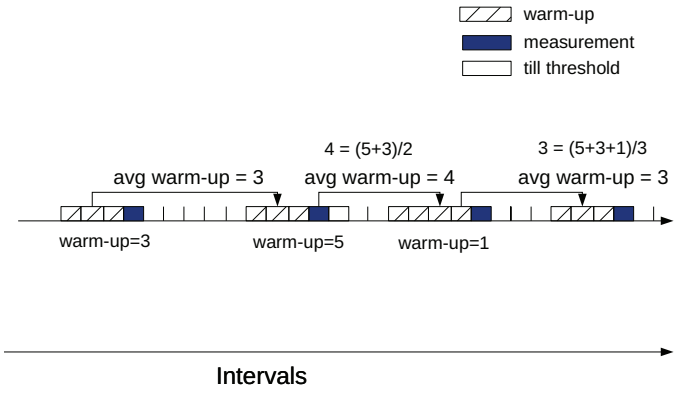


Figure 4.4: *Computing the average warm-up size.*

We found this approach to be robust and to bring the benefits of both worlds: the almost constant warm-up size avoids to bias the selection of performance measurement intervals, but the warm up size does depend on (automatically adapts to) the size of SRAM structures.

4.3.3 Rolling Window

In theory, the aforementioned warm-up threshold, used to decide when the SRAM structures are warm, is potentially architecture-dependent. Thus it should normally be exposed to the user; however, we implemented a safeguard which allows to use a fixed threshold whatever the size and number of SRAM structures.

The safeguard is that the fraction of warmed accesses is not computed based on all intervals since warm-up started but based on a rolling window of intervals. When we start our warm-up, all first accesses to the SRAM structure fields are

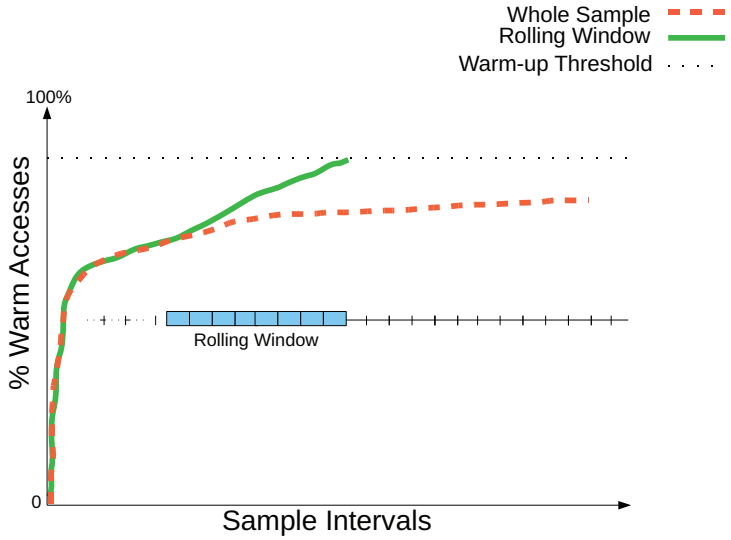


Figure 4.5: *Warm-up with rolling window.*

cold accesses. However, the majority of these cold accesses will be concentrated in the intervals occurring at beginning of our warm-up. Therefore if we calculate our percentage of warm memory accesses since the beginning of the warm-up period, they'll include these initial cold accesses and we might take a long time to achieve our warm-up threshold. Therefore instead of calculating the percentage of warm accesses since the beginning of warm-up period, we calculate them over this rolling window of latest intervals. In Figure 4.5, we see that, when the number of performance simulated intervals is equal to the size of the rolling window, i.e., the window starts rolling leaving the initial intervals behind, the fraction of warm accesses based on the rolling window intervals starts to increase faster than the fraction based on all intervals since the beginning of the sample. In other words, the large fraction of cold accesses in the first few intervals has no impact after the rolling window has shifted, and ultimately the fraction of warm accesses in the rolling window converges more quickly to 100%.

4.4 Methodology

Processor	SimpleScalar, out-of-order, 4 way
RUU/LSQ Size	16/8
Pipeline	5 stages
D-Cache/I-Cache	64KB, 2-way 32-byte block
Cache/Memory Latency	2/60 cycles
Memory Ports	2
Registers	32 Int, 32 FP
Functional Units	4 I-ALU, 1 I-Mult 4 F-ALU, 1 F-Mult 4 F-ALU, 1 F-Mult
I-TLB	16 4KB 4-way assoc blocks:lru
D-TLB	32 4KB 4-way assoc blocks:lru
Branch miss penalty	3 cycles miss lat

Table 4.1: *Simulator configuration.*

We used the SimpleScalar [12] 4-way out-of-order processor using the Alpha ISA, see Table 4.1. We modified the architectural structures to render it compatible with the TSE, by calling our SRAM library (which implements one bit for each line of the cache structures).

We used the SPEC2000 [39] benchmark suite to evaluate our sampling technique. Both SPECINT and SPECFP programs were used with *ref* input sets. In order to demonstrate the resilience of our technique to architectural changes, especially SRAM structure sizes, we vary the cache sizes from 8KB to 128KB. Detailed simulations of the full benchmark suite (no sampling) are used to obtain the actual CPI (Cycles Per Instruction) of the programs. The benchmarks used and some of their characteristics are listed in Table 4.2. We can see that the benchmarks have a range of characteristics, i.e., ones with high miss rate (`mcf, art`) to those with lower ones (`sixtrack, gap, mesa`). We also note that the miss rate decreases with the increase in cache size.

Later we also implement and test this mechanism in a PowerPC405 4.1 simulator running the MiBench suite (shown in Table 4.3) to test how it fares on a different architecture.

Program	# intervals	miss rate 8k	miss rate 64k	miss rate 128k
bzip	1.08878e+07	0.0339	0.0214	0.0188
crafty	1.91883e+07	0.0922	0.007	0.003
eon	8.06141e+06	0.03	0.0008	0.000
gap	2.69036e+07	0.0276	0.0165	0.0155
gcc.166	4.69177e+06	0.0829	0.073	0.0721
gcc.200	1.08626e+07	0.0622	0.04	0.0329
gzip	8.43674e+06	0.1118	0.0307	0.0091
mcf	6.18675e+06	0.4109	0.3927	0.3845
perlbmk	3.99293e+06	0.0322	0.0058	0.002
twolf	3.46485e+07	0.1082	0.0793	0.0701
vortex	1.18972e+07	0.0263	0.0106	0.0072
vpr	8.40688e+06	0.0783	0.0442	0.0378
ammp	3.26549e+07	0.0898	0.0641	0.0535
applu	2.23884e+07	0.139	0.1106	0.109
apsi	3.47923e+07	0.0834	0.044	0.0405
art	4.17954e+06	0.4024	0.4007	0.3993
equake	1.31519e+07	0.1837	0.1529	0.1484
facerec	2.11027e+07	0.0462	0.0377	0.0376
fma3d	2.68368e+07	0.073	0.0414	0.0408
galgel	4.09355e+07	0.1631	0.057	0.0549
lucas	1.42399e+07	0.1661	0.165	0.1649
mesa	2.81691e+07	0.0197	0.0047	0.0036
mgrid	4.19156e+07	0.1691	0.0679	0.0667
sixtrack	4.70949e+07	0.0142	0.0041	0.0041
swim	2.25831e+07	0.173	0.1729	0.1585
wupwise	3.49624e+07	0.0345	0.0278	0.0271

Table 4.2: *Benchmarks characteristics (SPEC2K).*

Unless otherwise stated, we use a *rolling window size* of 100 and 10 for the SPEC and MiBench programs respectively, and target a detailed simulation percentage of 1%.

4.5 Experimental Results (SPEC2K)

In the previous sections we detailed our interval selection mechanism as well as our warm-up methodology and how we were able to combine them together. In this section we present the results when we put our transparent statistical sampling technique into practice. In this section we focus only on the SPEC2K benchmark suite and detail the results for MiBench in the next one. The experiments have been

Program	# intervals	miss rate 4k	miss rate 8k	miss rate 64k
basicmath	83030	0.005407090	0.000740895	0.000005899
bfdec	20268	0.006585790	0.000201995	0.000000546
bfenc	20300	0.006585790	0.000201993	0.000000545
crc	68923	0.000909036	0.000000071	0.000000070
fft	75196	0.001069770	0.000499319	0.000236816
fft_inv	74185	0.000941445	0.000526624	0.000251666
mad	9562	0.014158300	0.003221000	0.000046522
patricia	10026	0.012848500	0.005001850	0.000404798
qsort	16104	0.003586100	0.002839120	0.001848140
rijndenc	3509	0.143746000	0.017091900	0.000003417
rijnddec	3399	0.147234000	0.020521900	0.000004706
susan_corner	421	0.004778040	0.002963450	0.002723390
susan_edge	1327	0.005752560	0.002422630	0.001926980
susan_smooth	77692	0.000808589	0.000312058	0.000009957
tiffbw	4119	0.016397900	0.015969600	0.000442368
tiffdither	19982	0.005737980	0.002974810	0.000007013
tiffmedian	14351	0.021464200	0.020735100	0.001540190
tiffrgba	3460	0.052524200	0.052445100	0.019427500
ispell 1	4550	0.010959900	0.005893440	0.003861380

Table 4.3: *Benchmarks characteristics (MiBench).*

conducted on the platform described in the last Section 4.4.

4.5.1 Simulation Time

Performance simulating whole benchmarks can be excruciatingly slow. It is orders of magnitude slower than the functional simulation and much more slower than running the benchmark on a real processor. A minute of execution time on a real processor can translate into weeks of performance simulation on the fastest of the simulators. But it's still a necessary evil as we cannot get the performance measurements without doing a detailed performance simulation. However we would like to keep the number of performance simulated instructions to a minimum as it translates directly into execution time.

In Transparent Sampling, we consider the fraction of total instructions to be simulated. The simulator starts running the program in functional mode. The simulation is partitioned in intervals of size N. At any interval, the TSE must

decide if the next interval will be performance simulated (sample collected) or just functionally simulated. For that purpose, it monitors the fraction of instructions performance simulated so far F_S, and the average warm-up size W_S (in number of intervals). Between two samples, as instructions are only functionally simulated, F_S decreases; when performance simulating instructions for a sample (warm-up and measurement), F_S increases.

Let F be the fraction of performance simulated instructions that the user requested. F can be interpreted as the probability that one interval should be performance simulated. However, when sampling is triggered, on average $W_S + 1$ intervals are performance simulated: W_S for warm-up + 1 for measurement. So, assuming a uniform distribution of samples, the probability that a sample (warm-up + measurement) is collected is $\frac{F}{(W_S+1)}$.

F is actually the *initial* probability that a sample is triggered at the next interval. As simulation progresses, the actual fraction of performance simulated instructions F_S will oscillate around F. If F_S exceeds F, the number of samples should be reduced, or conversely can be increased if F_S is less than F. Therefore, we use $\frac{F}{(W_S+1)}$ as the probability to sample at the next interval. W_S is adjusted as a function of F_S and hence controls the simulation probability. Since the user-specified bound is statistically enforced, the resulting number of performance simulated instructions will not exactly match the bound, but we found that this criterion allows to fall reasonably close in all cases.

Note that this sampling selection criterion is robust in spite of adaptive warm-up. If the SRAM structures are large, W_S will increase, as a result $\frac{F}{(W_S+1)}$ will decrease resulting in fewer samples. Conversely, if the warm-up requirements are low, the number of samples will be increased which will have a positive effect on accuracy, while remaining within the simulation time bounds set by the user.

The percentage of performance simulated instructions is indicated in Figure 4.6.

Even though the TSE uses a statistical control mechanism, the percentage is successfully maintained below 1.12% for all programs.

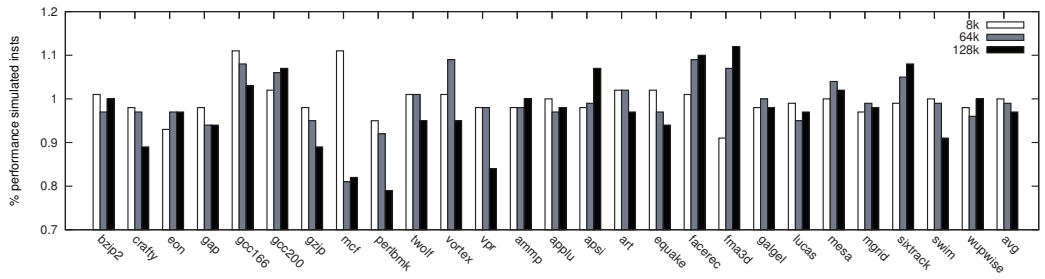


Figure 4.6: *% of performance simulated instructions (8KB, 64KB and 128KB caches).*

4.5.2 Warm-up length

We see in Figure 4.7 that as the size of SRAM structures increases the amount of detailed simulation required per sample also increases. The figure shows the systematic increase in the warm-up requirements of the architecture as the cache size is increased from 8KB to 128KB. This strengthens our argument that the warm-up needs to be adapted at run time to the architecture/program needs.

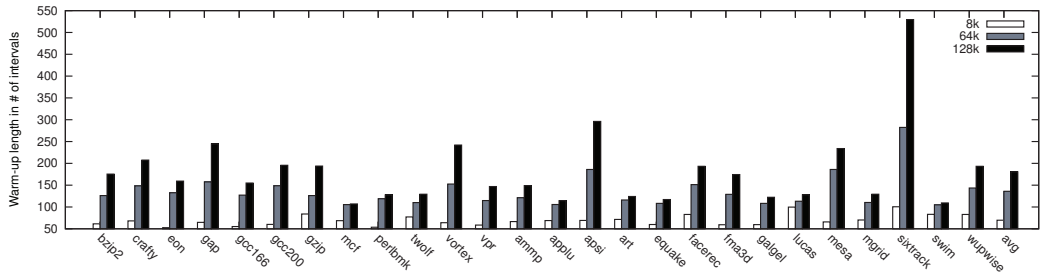


Figure 4.7: *Change in average warm-up length (8KB, 64KB and 128KB caches).*

4.5.3 CPI Error

The CPI error is indicated in Figure 4.8 for all benchmarks and the three aforementioned cache sizes. We first note the low absolute average error, less than 2%, which

is on par with the best sampling accuracy results [92, 110]. Moreover, this accuracy is stable across all three cache sizes, and does not increase with the cache size: the accuracy is 0.91% on average for 8KB, 1.47% for 64KB and 1.93% for 128KB. In other words, the accuracy does not change much as the SRAM structure size increases. To our knowledge, this is the first demonstration of an adaptive sampling strategy that exhibits stable accuracy as architecture characteristics change.

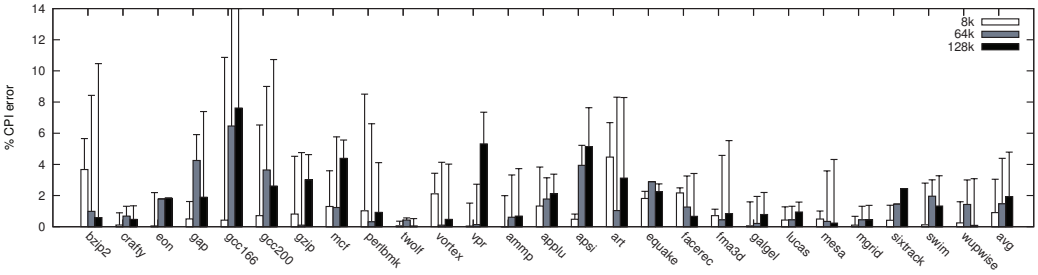


Figure 4.8: *CPI error (8KB, 64KB and 128KB caches).*

The solid bars in the figure show the percentage CPI error. This is the percentage difference between the CPI reported by the Transparent Sampling Engine and the CPI for the same program when it is simulated entirely in detailed mode.

From the solid bars we see that even though the overall error is low, some programs such as `vpr`, `apsi` and `gcc.166`, have errors up to 7%. Note that some of these programs are almost the smallest (in numbers of instructions) in their categories (`gcc.166` and `vpr`). If a program has significant warm-up requirements and the user has set a fixed percentage of simulation constraint, the TSE will adapt by reducing the number of samples to accommodate for the longer warm-up. If a program is small, the number of samples can become low enough to create a noticeable statistical variability.

The lines on top of the solid error bars represent the *Statistically Estimated Error* as described in the Section 4.2.1. This bar which represents the size of the confidence interval provides an estimate of the reliability of our measure. The confidence

interval indicates the interval of the estimated CPI. The upper bound of the confidence interval thus represents the estimated maximum CPI error. In Figure 4.8, one can notice there is no line (corresponding to the confidence interval) on top of some bars, for example `eon`. This occurs because either the real CPI is close to the upper bound of the confidence interval, or because it is even beyond it. While the former case is just the maximum error, the latter case can occur because the confidence interval can only be *estimated* as explained in Section 4.2.1.

One good news is the observation that wherever the actual CPI error is high, it is bounded by the Statistically Estimated Error. Since this error is reported to the user at the end of the simulation, the user can have an idea about the correctness of his result. A very large Statistically Estimated Error probably means that the actual error is also quite high and the user needs to simulate again perhaps with increased percentage simulation to have an accurate view. This is explained in the next section.

4.5.4 Bounding the Error

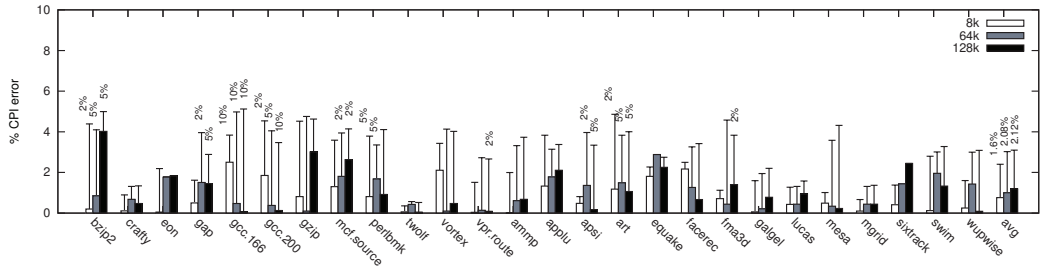


Figure 4.9: *TS after iterating, with 5% error target (% CPI error).*

While it is likely that a user would start with a low percentage of performance simulated instructions (time) objective, the user may also want to set accuracy objectives if they are not met by the initial simulation percentage. By taking advantage of the estimated confidence interval, a user can iteratively increase the percentage

of performance simulated instructions until the desired time/accuracy trade-off is reached. We illustrate this process below by setting a target of 5% maximum error for the Statically Estimated Error (10% confidence interval size), and apply this iterative process to all benchmarks whose confidence interval with 1% simulation was greater than 10%.

For each cache benchmark pair, we individually increase the percentage of performance simulated instructions to 2%, 5% and 10% until our accuracy goal is reached. In Figure 4.9, we show the resulting error and confidence interval, using 1% performance simulated instructions and when applying the iterative process; the target fraction of instructions used is indicated on top of each bar when it is different from 1%. We see that for `gcc.166` `mcf` and `vpr`, with a simulation of 10%, 2% and 2% respectively the confidence intervals are significantly reduced and the accuracy has improved as well.

4.5.5 Fixed Warm-up

As mentioned in Section **??**, the main other alternative to adaptive warm-up, requiring no functional simulator modification, and compatible with frequent target program modifications, would be to use a fixed warm-up combined with sampling. In Figures 4.10 and 4.11, we evaluate three fixed warm-up sizes: 10, 100 and 1000 intervals. The x-axis of the figures show the average values first for groups of *fp* and *int* benchmarks and then the combined average for all three cache sizes. While, for any architecture, there exists a sweet spot where the fixed warm-up size would realize a good accuracy/simulation time trade-off, such as 100 intervals in this case, the approach is not robust. As the architecture characteristics change (e.g., larger caches), the warm-up size can become underestimated and result in large CPI error. As the 10-interval warm-up results show in Figure 4.10: it performs well for small caches (8KB), but poorly for medium and large caches (64KB and 128KB).

Conversely, the warm-up size can be overestimated, and significantly increase the simulation time, as the 1000-interval warm-up results show in Figure 4.11.

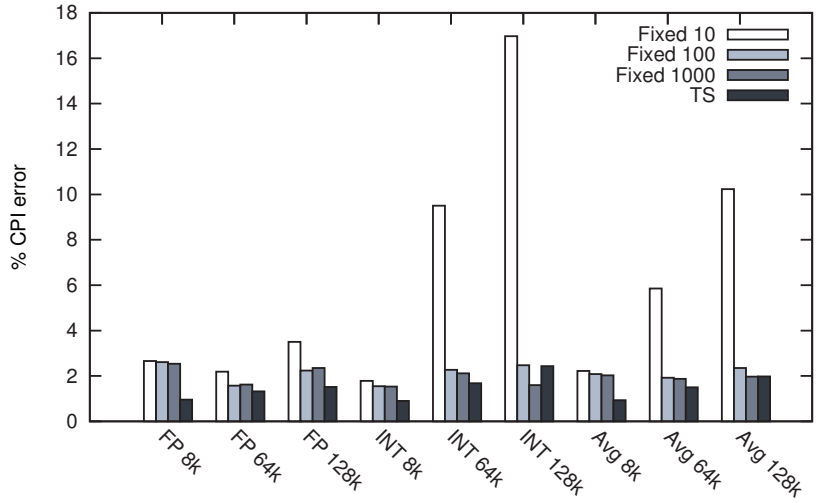


Figure 4.10: *TS vs. fixed warm-up (% CPI error).*

4.5.6 Interval Size

Figures 4.12 and 4.13 respectively show the CPI error and the warm-up length in number of instructions when varying the interval size from 1,000 instructions to 1,000,000 instructions for 64KB caches. The counter-intuitive best choice is to use a small interval size of 10,000 or 100,000 instructions (we use 10,000 instructions throughout this study). It is counter-intuitive because the smaller the intervals, the higher the fraction of performance simulated instructions used for warm-up rather than measurement, since the warm-up size is constrained by the SRAM structures. However, small intervals enable to achieve higher accuracy by multiplying the number of samples. This trade-off underlines that it is necessary to dedicate most of the performance simulation time to warm-up rather than measurement. Note that the 1,000-instruction intervals provide no significant benefit in simulation time at a noticeable cost in accuracy. While the results are only shown for 64KB caches, they

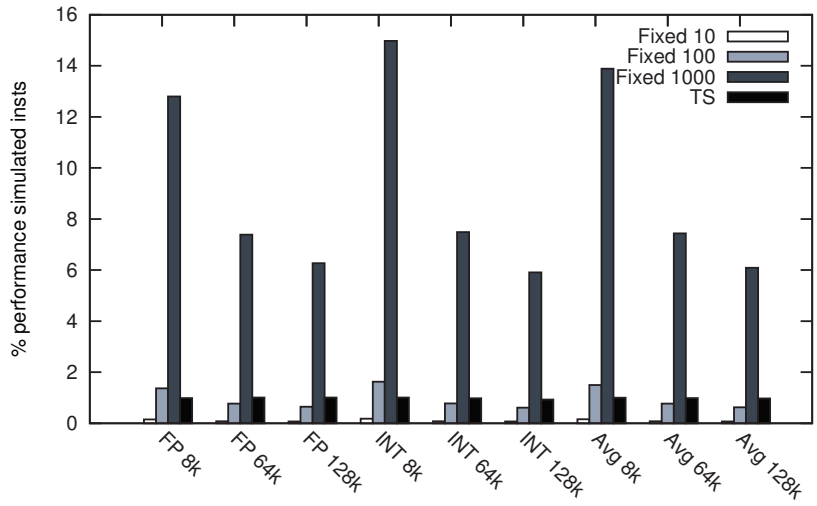


Figure 4.11: *TS vs. fixed warm-up (% performance simulated instructions).*

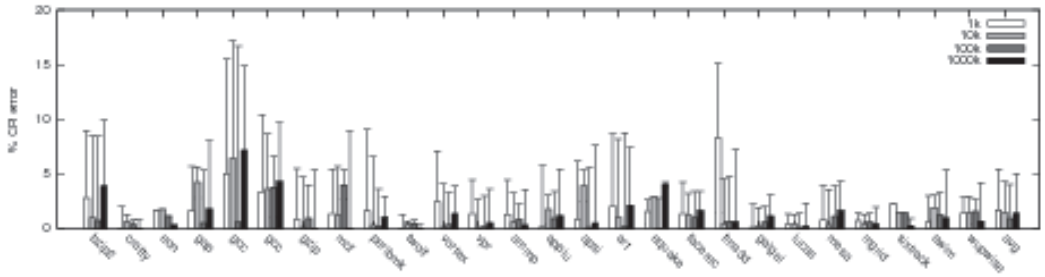


Figure 4.12: *Impact of interval size (% CPI error).*

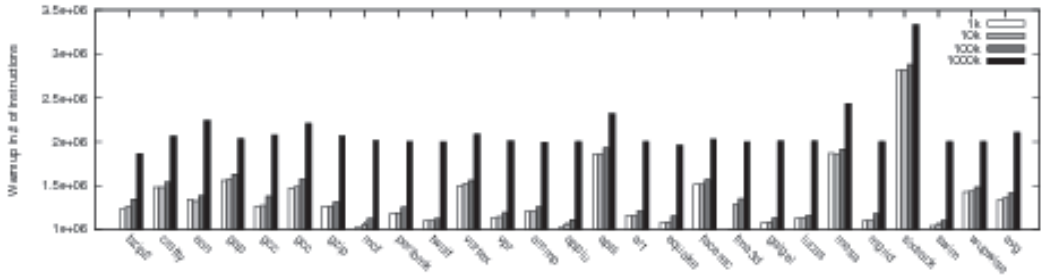


Figure 4.13: *Impact of interval size (% performance simulated instructions).*

are consistent across all cache sizes and benchmarks.

4.6 Experimental Results (MiBench)

As described earlier, one of our goals was to develop a robust technique which performs well on all architectures. To this intent, Figures 4.14 and 4.15 show the result of applying the statistical representative sampling to MiBench programs on the PPC405 simulator. The first thing to point out from Figure 4.14 is that Transparent Statistical Sampling does a pretty good job of estimating the program performance. This can be seen in the low average error rates (2.96%, 1.67%, and 1.62% for cache size of 4KB, 8KB, and 64KB respectively). A couple of benchmarks `susan_edge` and `tiffrgba` show high performance estimation errors (17% and 20% for 4KB cache respectively). When we get down into details to investigate the matter, we see that the small lengths of the MiBench benchmarks play an important role. MiBench programs are more than 100x smaller than the SPEC2K ones on average. Since we try to limit the percentage of detailed simulation to within 1% of the program size, we simulate only a small amount in detail. Therefore we manage to collect only a few samples for these small-sized benchmarks. Table 4.4 shows this by listing the total number of intervals for each program as well as the number of samples collected for each cache size. We see that for very few benchmarks the number of samples is more than 100 and for some it is even in single digits. Such is the case for `susan_edge` and `tiffrgba` with 2 and 8 samples respectively. Having very few samples can not only provided a distorted measure for program performance but also it renders the statistical confidence interval calculations (Statistically Estimated Error) meaningless. As discussed at the end of section 4.2.1 and shown in Figure 4.3, having too few measurements to calculate the confidence interval does not satisfy the normalcy assumptions of the Central Limit Theorem and therefore can give erroneous resluts. This the reason we see absurdly high confidence intervals for `susan_edge` and `tiffrgba`. The statistically estimated error is 75% for `susan_edge` for 4KB cache and 52.5% and 46% for `tiffrgba` in 4KB and 8KB cache configurations re-

Program	intervals	4k_samples	8k_samples	64k_samples
basicmath	83030	193	133	119
bfdec	20268	48	35	42
bfenc	20300	29	28	35
crc	68923	232	232	235
fft	75196	238	194	150
fft_inv	74185	238	186	122
mad	9562	21	14	3
patricia	10026	23	10	9
qsort	16104	43	34	26
rijndenc	3509	9	6	8
rijnddec	3399	5	3	6
susan_corner	421	1	1	2
susan_edge	1327	2	2	2
susan_smooth	77692	216	186	198
tiffbw	4119	11	7	5
tiffdither	19982	49	31	14
tiffmedian	14351	33	24	14
tiffrgba	3460	8	5	6
ispell	14550	48	24	12
avg	27389.68	76.16	60.79	53.05

Table 4.4: *Number of samples for statistical sampling on MiBench.*

spectively. Similarly susan_corner has a confidence interval of size 0, not because it estimates the program performance exaclty but because it has only one sample.

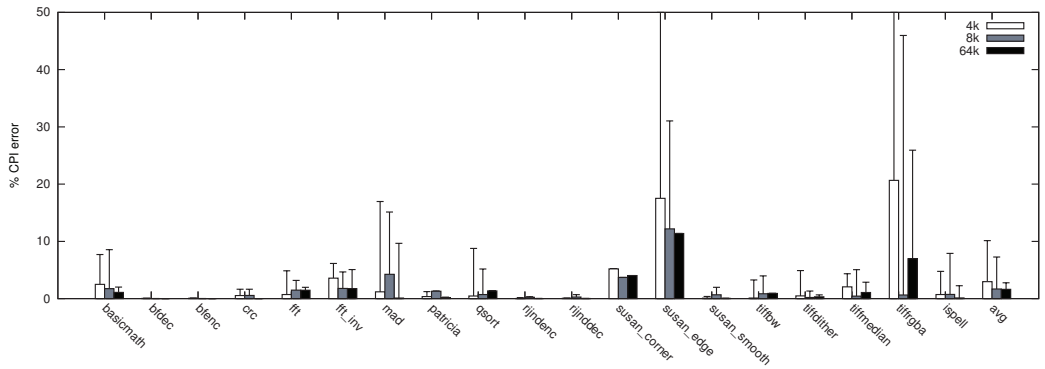


Figure 4.14: *CPI error (4KB, 8KB and 64KB caches).*

Nevertheless we try to iteratively increase the percentage of performance simulated instructions hoping to lower the statistically estimated error below 5% as we did in the previous section. Figure 4.16 shows the result. Though we succeed in most of the cases, we can still see that for susan_edge and tiffrgba, even after

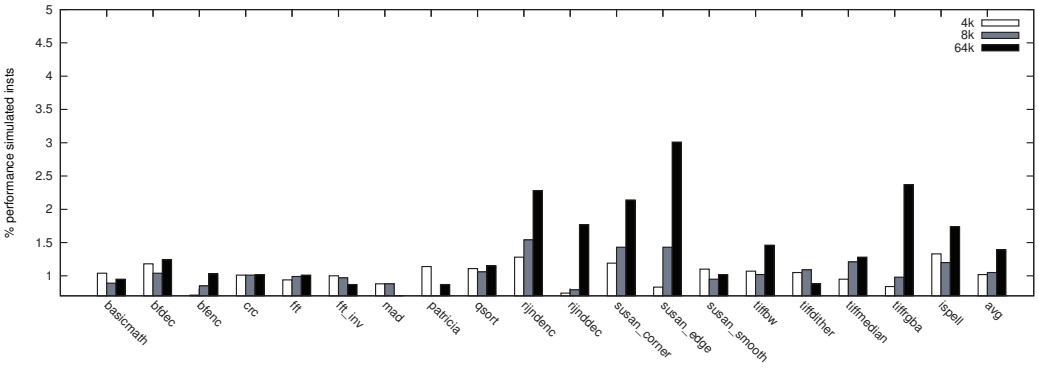


Figure 4.15: *% of performance simulated instructions (4KB, 8KB and 64KB caches).*

having simulated 30% of the program in detail, we could only reduce the statistically estimated error to 10%.

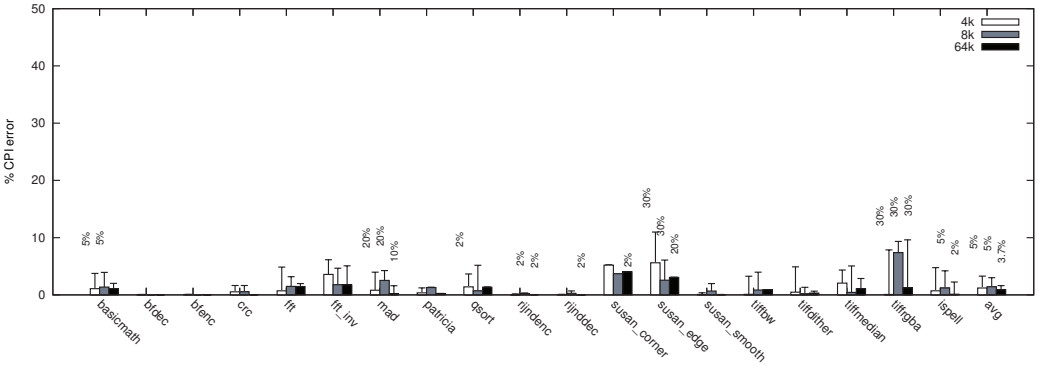


Figure 4.16: *CPI error after iterating (4KB, 8KB and 64KB caches).*

4.7 Warm-up Parameters

We discussed in Section 4.3.2 the shifting effect of SMA style adaptive warm-up and how it biases the selection of performance simulated intervals towards certain portions of the program. We also described how we had to devise the method of using average warm-up length for measurements in order to avoid this. We now show the effect of this biased selection on the CPI error.

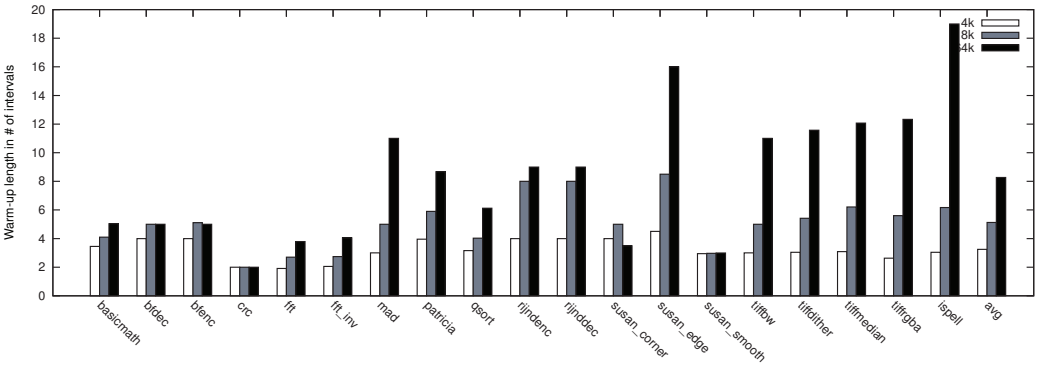


Figure 4.17: *Average warm-up length (4KB, 8KB and 64KB caches).*

The Figure 4.18 shows the CPI error for different benchmarks for all the three cache sizes when we use only SMA as our warm-up criterion without the average warm-up length. We can see that while it works for most of the benchmarks, `gcc.166` shows an error of about 13% for a 64k cache and `art` exhibits an enormous error of 25% for an 8k cache. These two cases justify our introduction of the average warm-up length.

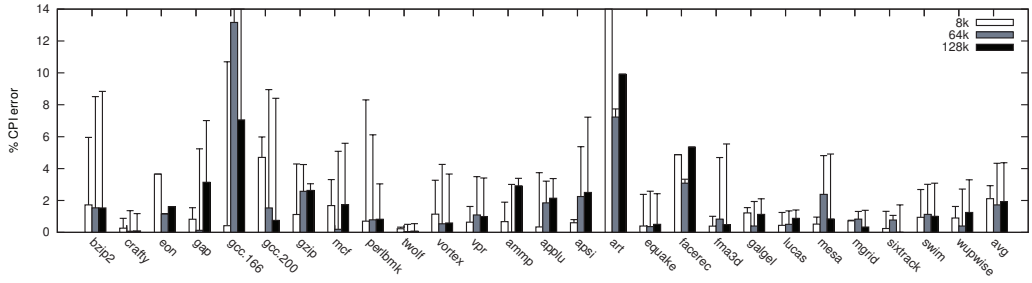


Figure 4.18: *TS using only warm-up threshold (% CPI error).*

Figure 4.19 shows that the percentage of detailed simulated instructions was still controlled to 1%.

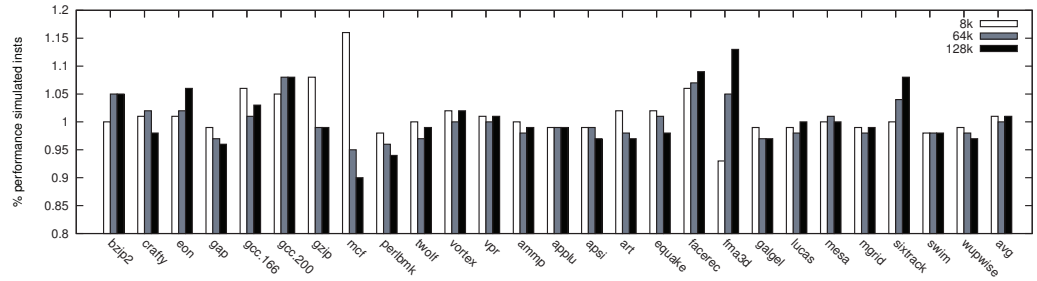


Figure 4.19: *TS using only warm-up threshold (% performance simulated instructions).*

4.7.1 Rolling Window Size

Section 4.3.3 describes why we need to calculate the percentage of warm accesses over a rolling window instead of since the beginning of warm-up. Achieving the warm-up threshold faster lets us spend less time in warm-up and more time gathering performance measurements when we are constrained by our simulation budget.

We needed to decide the length of this rolling window and decided to settle for a rolling window of 100 intervals. Figures 4.20 and 4.21 show the effect of varying the rolling window size on the percentage CPI error and the average warm-up length respectively.

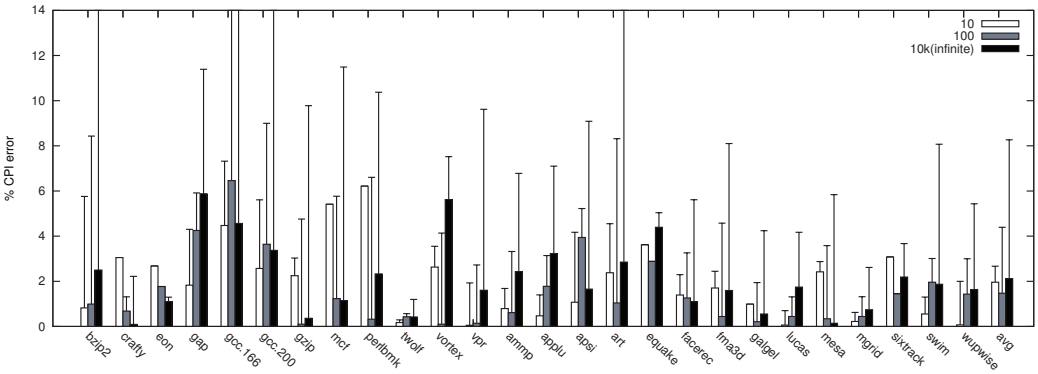


Figure 4.20: *Varying the size of Rolling Window (Error).*

The Figure 4.21 shows the change in warm-up length for the benchmarks as we vary the rolling window size from 10 and 100 intervals to infinite. We can clearly see

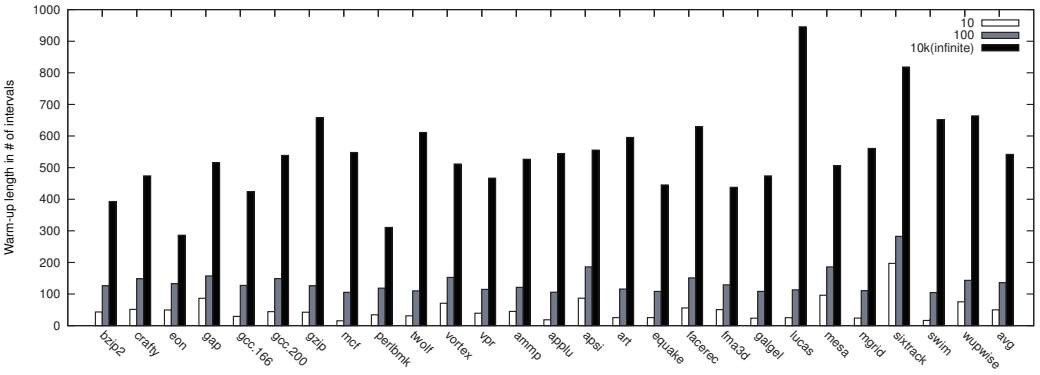


Figure 4.21: *Varying the size of Rolling Window (Warm-up length).*

that in the absence of a rolling window (infinite case), the effect of initial cold misses is extremely pronounced on the warm-up length. This is demonstrated by the long bars showing much longer average warm-up lengths compared to the rolling window. Indeed, the average *average warm-up length* is four time longer (542 intervals) than when using a rolling window of 100 (134 intervals). This results in a corresponding reduction in the number of samples and a higher error, as shown in the figure 4.20. One good news is that, though the increase in average warm-up length and the corresponding decrease in number of samples is manifold, the change in average performance error is not that great. This means that even with a rolling window of 100 intervals we should be able to achieve the same performance with fewer samples.

Though the smaller rolling window of 10 intervals results in low average warm-up lengths and hence more number of samples, we found that, in many cases, it was still sensitive to variations in warm-up linked to program regions and caused more error than a rolling window of 100 intervals.

4.7.2 Warm-up Threshold

In Section 4.3.3 we indicate that we need to attain a threshold of percentage warm accesses before we take our measurement. One criterion we considered was: when do

we stop the warm-up and take the performance measurement? The obvious answer is: when the microarchitecture is completely warm and all accesses done by the program are warm accesses. The problem with this approach is that it takes much longer to achieve a complete warm-up then it takes to achieve an *almost complete* warm-up.

We need to know when the warm-up is finished in order to calculate the average warm-up length. We had to choose a warm-up threshold, to be attained each time we take a sample, such that it eliminated most of the bias caused by cold misses and did not cause too much performance simulation. To that purpose, we tried different warm-up thresholds. Figure 4.22 shows how the CPI error percentage for warm-up thresholds of 95%, 99.9% and 100%. As can be seen 95% of warm-up is not enough in most of the cases and causes an average error of 14% over all benchmarks. What was surprising was that a warm-up of 100% performs less than that of 99.9%. This is explained by the fact that the simulator has to simulate for a long time each time it tries to achieve the 100% threshold. This results in a longer average warm-up length. This can be seen in Figure 4.23 where a warm-up threshold of 100% causes the average warm-up length to treble as compared to that of 99.9%. Since we have a mechanism in place to regulate the number of performance simulated instructions is mentioned in Section 4.5.1, this results in fewer samples and thus the accuracy of the result is affected. The most flagrant example is `perlbmk` whose average warm-up length increases tenfold resulting in a corresponding 10x decrease in the number of samples and a corresponding increase in performance estimation error.

4.8 Conclusion

In this chapter, we present *Transparent Statistical Sampling*, a sampling technique that reconciles sampling and warm-up techniques by delivering state-of-the-art accuracy and simulation time, while remaining easily accessible to end users.

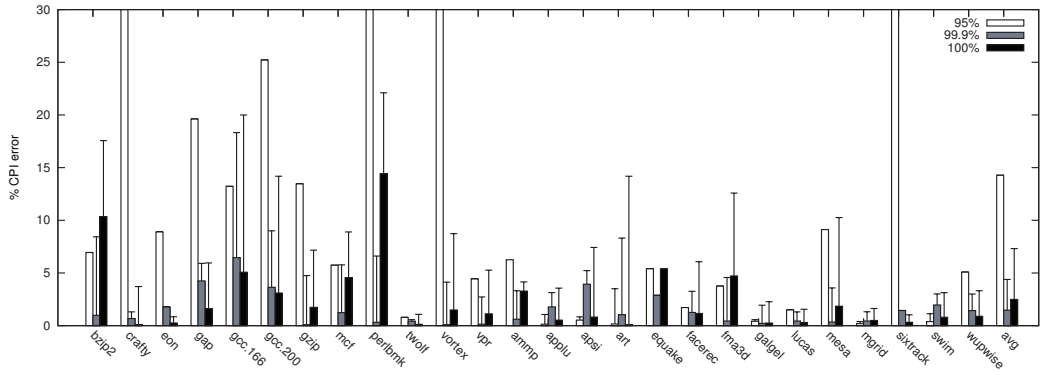


Figure 4.22: *Effect of varying warm-up threshold (Error).*

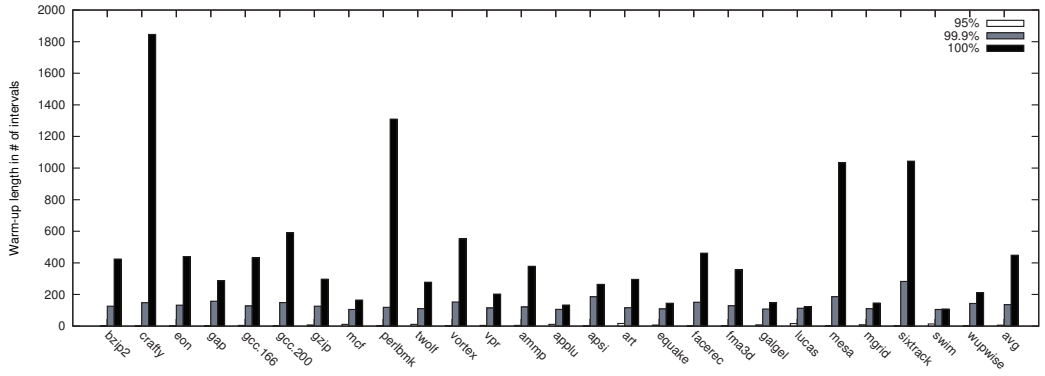


Figure 4.23: *Effect of varying warm-up threshold (Warm-up length).*

The contributions of this work can be viewed from the following angles. One, it proposes to combine the statistical sampling with the latest warm-up methodology. Secondly, in combining SMA with statistical sampling, it discovers the bias introduced by the adaptive warm-up in the selection of intervals and proposes a method to bypass this defect.

Transparent Statistical Sampling achieves a CPI error of 1.47% with 20.7 million performance simulated instructions (1%) on average for 64K caches.

We demonstrated the resilience of this technique to program and architecture changes by testing it on SimpleScalar/SPEC2K and PPC405/MiBench combinations for three different cache sizes each. Though the results for MiBench programs are

slightly inferior than those for SPEC2K program, we believe that the small lengths of the MiBench programs are to blame. They do not run long enough to give us time to collect enough number of samples to do meaningful statistical calculations.

Encouraged by the results, we intend to further this work by testing its applicability on different other microarchitectural structures like branch predictors, TLBs and multi-level caches. Also, we'd like to explore the working of this technique in the context of multi-core architectures.

Chapter 5

Conclusion

In this chapter we recapitulate and conclude the discussion proffered in the preceding chapters and provide directions for future extensions in which this work can be extended.

5.1 Summing it up

As shown in previous literature and demonstrated by the discussion in this document, we conclude that Sampling indeed is an effective technique in that it manages to reduce simulation times with a minimum loss in the accuracy of the results.

Both representative and statistical sampling techniques have been shown to provide acceptable results, though the latter more so.

Representative Sampling. While the representative sampling analyzes the code to intelligently select samples from the program execution, it requires to do a pass on the executed instruction stream to identify the representative portions of the program. Attempts to do online phase classification have been less successful as they do not have the whole picture in front of them when clustering program intervals. This results in greater than optimal number of clusters and correspondingly increased detailed simulation which directly affects simulation time. The online im-

plementation limitations of representative sampling encourages the need to explore other techniques, like statistical sampling.

Statistical Sampling. The ability of statistical sampling to not to require a pre-analysis makes it a strong candidate for online implementation of sampling. Without any prior knowledge of the characteristics of program execution, in this type of sampling, intervals are selected randomly for detailed simulation. Each interval has an equal probability of selection. Another advantage of this approach is that it can rely on statistical theory to provide confidence estimates in the reliability of the results. Furthermore, these statistical measures can be tweaked to achieve a desired compromise between performance accuracy and simulation times. In our experiments we found the statistical sampling technique to deliver superior performance when compared to an online implementation of representative sampling.

Phase Prediction. While attempting to implement an online representative sampling technique, we experimented with online phase classification and prediction. We noted that the quality of online phase characterization was inferior to that of its off-line counterpart. This was expected as the off-line analysis entails analysis of the whole program instruction stream before it starts classification whereas an online method has visibility only until the present moment. As this online classification produces too many phases, this also affects the phase prediction mechanism. Phase prediction is needed in order to prepare in advance (warm-up) for the desired interval to sample. A large number of phases detected would result in more combinatorial sequences of phases and thus make them harder to predict. Thus we were forced to use heuristics, such as to cut off phases with negligible weight, to decrease the number of phases and improve their predictability. Secondly, as we need to predict a phase many intervals before it actually occurs, we need to guess the IDs of multiple consecutive future phases. The prediction accuracy decreases as we try to predict more and more farther than our current location. Low prediction accuracies can

severely degrade the simulation times as we would be simulating intervals in detail when we should not be.

Warm-up. Warm-up, achieving the correct simulator state before sampling, is still a relevant issue. Especially for small-sized intervals, we observed that not having the correct micro-architectural state can skew the performance measurements to a degree which is unacceptable. Despite recently developed good warm-up techniques (MRRL, BLRL, SMA), not all of them let themselves adaptable to online sampling techniques easily. We, ourselves, could only find SMA to suite our requirements. An often cited argument against the relevance of warm-up is that choosing sufficiently large intervals obliterates the need for warm-up at all. The case for large intervals is weakened by the observation that due to their size there has to be only a few of them in order to restrain the simulation time. This low number of simulated intervals renders the statistically calculated confidences meaningless. Therefore, instead of random selection, intervals have to be chosen intelligently for detailed simulation. This often requires a pre-analysis of program instruction stream to identify the most suitable candidates. Secondly, the effect of warm-up depends upon the architectural parameters and it is not clear how large an interval should be to nullify warm-up bias for a particular architecture. Another type of warm-up, called functional warm-up, as we mentioned previously, exposes the user to implementation details. We proposed an adaptive warm-up strategy which adjusts the detailed simulation needed for warm-up dynamically as a function of program/architecture needs.

Usability. In designing our sampling approach, the ease of use for the end user was one of our prime concerns. Figuring out parameter values for each hardware/-software configuration exposes the end user to implementation details. Thus, for the architects already overwhelmed by design problems, this added complexity of using a simulation technique may prove a detriment to technique adoption. Keeping in view this requirement, we designed a sampling mechanism which adapts to

a given hardware/software configuration and needs minimal user intervention. The only modification to the simulator is addition of a bit to each SRAM entry. To this end we provide a class library from which the user might inherit his structures and thats all what is needed to plug in. This one class would suffice for all SRAM based structures.

5.2 Future Directions

We propose an adaptive sampling strategy to reduce simulation times. While we have demonstrated its effectiveness in some limited scenarios, it is to be seen how far this approach can be extended. Below we discuss a few areas in which this work could be complemented and extended.

Effect of independent events. When simulating a benchmark on a simulator, sampling can effectively capture the program performance. However, a simulator may model hardware events which affect performance but which the sampling technique may not be conscious of. Examples may include DRAM refreshes which may delay the completion of memory requests or the processor being interrupted by a communicating peripheral. These events affect the program runtime but may not be captured by sampling. Representative sampling, which analyzes the executed code to identify representative intervals for simulation, is especially affected because these events occur independent of the program. The low number of intervals simulated in detail may miss these instances. Similarly for systematic sampling, if the time period of the sampling is a multiple of that of these events, it may sample too many of such events or miss them altogether thus under- or overstating the IPC. Wunderlich *et al.* [110] claim having verified that there is no such regular repetition in program characteristics but they do not take into account such events which are transparent to the executing code. In our simulations we observed that DRAM refreshes occur at regular intervals and can affect performance. We believe that random sampling,

if used in such scenarios, should be an effective solution.

A related issue is that often simulated benchmarks are tested in isolation, i.e., only the benchmark is executing on the simulated processor. This is in contrast to real life scenarios where the application shares the processor with other applications and the operating system. The effect of the operating system environments [21] on application performance has been documented. We intend to extend our sampling approach to full-system simulation to quantify how it performs in an environment where the applications are continuously interrupting each other and also being interrupted by peripherals.

Multicore Sampling. Computer industry has chosen the multicore road for its future, for better or for worse remains to be seen. With the advent of multicore chips, the process of porting the software to these parallel architectures has already begun. We'll be seeing more and more of parallel programs and benchmarks. The original roadmap of this thesis, unachieved due to slow progress and time constraints, included devising sampling strategies of multicore simulations. Sampling is easier to implement on single core processors because it is simple to fastforward the program execution from one point to another. Indeed, the strength of sampling lies in its ability to execute most of the program in fastforward mode. Multicore chips executing many processes/threads in parallel bring non-determinism into the equation. In detailed cycle accurate mode, different portions of concurrent threads can overlap with each other in different ways causing different access pattern contentions and result in different performances. A slight difference in the ordering of instructions can result in another thread taking hold of the mutex/lock first, changing the entire flow of execution after that point. This may completely alter the performance seen. During functional fastforward mode, a simulator has no idea which instructions of a thread would overlap in execution with which ones of another determining their relative execution speeds. Thus it does not know how much to fastforward each thread

relative to others. This proposes an interesting problem for the implementation of sampling in multicore simulators.

Previously, there have been attempts [102] to apply representative sampling to multi-context processors. The proposed co-phase scheme identifies phases individually in each thread and then tries to sample all co-occurring phase combinations among threads. It is a very limited approach. As the number of threads increases, the number of co-phases explodes. This makes trying to simulate each co-phase combination an unrealistic option. Furthermore, this approach assumes the same behaviour for each occurrence of a co-phase, however a different ordering of instructions within a co-phase can produce different results.

SimFlex [107] attempts the statistical sampling of parallel workloads. They attempt to randomly sample transactions and concentrate only on the user-mode instructions as their retirement rate shows less variation than the actual transaction completion rate. By ignoring the system part of the execution they are completely cutting off the effects of the operating system execution which can have a considerable effect on the performance of the system.

We believe that multicore simulation presents an opportunity to test and demonstrate the usefulness of statistical sampling techniques. Enough randomly simulated samples should be able to capture all interleavings of the instructions of different processes executing in parallel. The sheer nature of the random coupling of threads and the interleavings of their instructions makes random sampling a good candidate. Using small sized intervals permits increasing the number of samples and capture fine-grained interactions between different threads. Increased number of samples can help provide good confidence in the performance measures. It needs to be seen whether we should stop simulating when a system-wide confidence interval has been achieved or we should wait for the measurements from each thread to attain a steady state.

For the fastforwarding part, there have been heuristics, i.e. fastforward each thread based on its average or most recent IPC. These can be combined with existing techniques like direct execution to fastforward the threads by executing them on the host processors. Fastforwarding using the host processors is much faster than fastforwarding in a simulator. Thus it may help alleviate multicore simulation times.

For the warm-up part, we believe that the SMA-style warm-up applied in this work shall work fine for multicore simulations as well. The same technique of adding a warm-up bit to SRAM entries can be extended across cores. A point of discussion may be whether each thread should have its own warm-up threshold, or a system-wide warm-up threshold be used, or a combination of the two. Again, exploration has to be done to verify the effect of each of these conditions.

Another approach which can help is hardware execution. FPGAs can be of assistance in fastforwarding the portions of program which are not needed for detailed simulations. Simulators are being implemented in FPGAs such as the RAMP project at University of Texas at Austin which implements a platform to simulate multi-processor set-ups. The flexibility of the FPGAs and their clock speeds make then an interesting alternative for multicore simulators, as simulator parallelization has yet to have a break through.

Simulation acceleration. Finally, the author believes that the simulation acceleration problem is a multifaceted one. The complexity of modern architectures and the size of programs render achieving accurate estimates of performance in reasonable time much difficult. Therefore, relying on one method of simulation acceleration might not be the right choice. Instead, it is recommended to combine the strengths of different techniques in a synergistic manner. For example, sampling is good but with the increasing sizes of benchmark programs, the bottleneck will soon be the execution of the functional portion of the program. This would need faster simulators. FPGAs are a good alternative. Though having a little steep learn-

ing curve, they should be able to provide much faster fastforwarding than software based simulators. Similarly, there are portions of simulated architectures that are not very complex to model. Using analytical modeling techniques to model relatively simple structures would also relieve some burden from the simulators. Using random sampling is also encouraged due to its simplicity of implementation and also because statistical confidence measures are a useful indication of the correctness of the sampling process. Such a hybrid approach which draws on the strengths of all these techniques can provide a useful way to keep up with the increasing difficulty of the simulation task.

Bibliography

[1] M. Annavaram, R. Rakvic, M. Polito, J. yves Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *In Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 93–104, 2004.

[2] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[3] I. ARM Holdings. Advanced risc machine, arm holdings, inc. http://www.arm.com/.

[4] D. August, J. Chang, S. Girbal, D. Gracia Pérez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *Computer Architecture Letters (CAL)*, 6(2):45–48, 2007.

[5] T. Austin. Position paper for the 2001 nsf workshop on computer performance evaluation techniques. *Computer*, 2001.

[6] M. aware Floorplanning, V. Nookala, D. J. Lilja, and S. S. Sapatnekar. Comparing simulation techniques for.

[7] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM.

[8] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257, New York, NY, USA, 2000. ACM.

[9] K. C. Barr, H. Pan, M. Zhang, and K. Asanovi. Accelerating multiprocessor simulation with a memory timestamp record. In *In ISPASS-2005*, pages 66–77, 2005.

[10] B. Black and J. Shen. Calibration of microprocessor performance models. *Computer*, 31(5):59 –65, may. 1998.

[11] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[12] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

[13] H. W. Cain. Precise and accurate processor simulation, 2002.

[14] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, G. Fursin, and O. T. O. Hipeac. Automatic performance model construction for the fast software exploration of new hardware designs. In *In ACM International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 24–34, 2006.

[15] D. Citron. Misspeculation: partial and misleading use of spec cpu2000 in computer architecture conferences. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 52–61, New York, NY, USA, 2003. ACM.

[16] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 468–477. IEEE Computer Society, 1996.

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

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

[19] P. Crowley and J.-L. Baer. On the use of trace sampling for architectural studies of desktop applications. *SIGMETRICS Perform. Eval. Rev.*, 27(1):208–209, 1999.

[20] S. Dasgupta. Experiments with random projection. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 143–151, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[21] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 266–277, New York, NY, USA, 2001. ACM.

[22] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. pages 217 – 227, dec. 2003.

[23] A. S. Dhodapkar and J. E. Smith. Managing multi-configurable hardware via dynamic working set analysis. In *In 29th Annual International Symposium on Computer Architecture*, pages 233–244, 2002.

[24] C. Dubach, T. Jones, and M. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 262–271, Washington, DC, USA, 2007. IEEE Computer Society.

[25] P. K. Dubey and R. Nair. Profile-driven generation of trace samples. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 217–224, Washington, DC, USA, 1996. IEEE Computer Society.

[26] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220, Washington, DC, USA, 2003. IEEE Computer Society.

[27] L. Eeckhout and K. De Bosschere. Efficient simulation of trace samples on parallel machines. *Parallel Comput.*, 30(3):317–335, 2004.

[28] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation. *The Computer Journal*, 48(4):451–459, 5 2005.

[29] L. Eeckhout, S. Nussbaum, J. Smith, and K. De Bosschere. Statistical simulation: adding efficiency to the computer designer's toolbox. *Micro, IEEE*, 23(5):26 – 38, sep. 2003.

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

[31] M. Ekman and P. Stenstrom. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *In Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 89–99, 2005.

[32] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[33] A. Falcon, P. Faraboschi, and D. Ortega. Combining simulation and virtualization through dynamic sampling. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:72–83, 2007.

[34] A. P. Falsafi, S. Chen, and S. Chen. Direct smarts: Accelerating microarchitectural simulation through direct execution. Technical report, 2004.

[35] D. Gracia Pérez, H. Berry, and O. Temam. Budgeted region sampling (BeeRS): do not separate sampling from warm-up, and then spend wisely your simulation budget. *International Symposium on Signal Processing and Information Technology*, 0:1–6, 2005.

[36] D. Gracia Pérez, H. Berry, and O. Temam. Iddca: A new clustering approach for sampling. In *International Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, 2005.

[37] D. Gracia Pérez, G. Mouchard, and O. Temam. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. In *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, pages 43–54. IEEE Computer Society, 2004.

[38] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, 2001.

[39] J. L. Henning. Spec cpu2000: Measuring CPU performance in the new millennium. *Computer*, 33:28–35, 2000.

[40] J. Hruska. Phantom phenom's perplexing processor problem behind product delay. *Ars Technica*, 3rd Dec. 2007.

[41] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 209–220, Washington, DC, USA, 2002. IEEE Computer Society.

[42] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. *SIGOPS Oper. Syst. Rev.*, 40(5):195–206, 2006.

[43] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. *SIGOPS Oper. Syst. Rev.*, 40(5):195–206, 2006.

[44] C. Isci and M. Martonosi. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. *High-Performance Computer Architecture, International Symposium on*, 0:121–132, 2006.

[45] R. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling / Raj Jain*. Wiley, New York :, 1991.

[46] B. Jenkins. Algorithm Alley: Hash Functions. http://www.ddj.com/184410284.

[47] J. John W. Haskins and K. Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. *ISPASS '05: IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.

[48] A. Joshi, Y. Luo, and L. John. Applying statistical sampling for fast and efficient simulation of commercial workloads. *Computers, IEEE Transactions on*, 56(11):1520 –1533, nov. 2007.

[49] D. A. C. Joshua Kihm. Cogs-sim: Co-phase guided small-sample simulation of multithreaded and multicore architectures, 2007.

[50] N. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *Computers, IEEE Transactions on*, 38(12):1645 –1658, dec. 1989.

[51] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. *SIGARCH Comput. Archit. News*, 32(2):338, 2004.

[52] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictivemodeling for cross-program design space exploration in multicore systems. pages 327 –338, sep. 2007.

[53] J. L. Kihm and D. A. Connors. Statistical simulation of multithreaded architectures. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:67–74, 2005.

[54] J. L. Kihm, S. D. Strom, and D. A. Connors. Phase-guided small-sample simulation. In *ISPASS*, pages 84–93. IEEE Computer Society, 2007.

[55] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, 2002.

[56] S. Kodakara, J. Kim, D. Lilja, D. Hawkins, W.-C. Hsu, and P.-C. Yew. Cim: A reliable metric for evaluating program phase classifications. *IEEE Comput. Archit. Lett.*, 6(1), 2007.

[57] V. Krishnan and J. Torrellas. A direct-execution framework for fast and accurate simulation of superscalar processors. In *In International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, 1998.

[58] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. pages 145–163, 2001.

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

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

[61] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 236–247, Washington, DC, USA, 2005. IEEE Computer Society.

[62] J. Lau, S. Schoenmackers, and B. Calder. Structures for Phase Classification. *ISPASS '04: IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[63] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification, 2004.

[64] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *In 11th International Symposium on High Performance Computer Architecture*, pages 278–289. IEEE Computer Society, 2005.

[65] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical report, Mountain View, CA, USA, 1993.

[66] P. S. Levy and S. Lemeshow. *Sampling of Populations: Methods and Applications*. Wiley, New York, 3 edition, 1999.

[67] D. J. Lillja. *Measuring Computer Performance : A Practitioner's Guide*. Cambridge University Press, New York, NY, 2000. ISBN 0-521-64105-5.

[68] W. Liu and M. C. Huang. EXPERT: expedited simulation exploiting program behavior repetition. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 126–135. ACM Press, 2004.

[69] Y. Luo, L. K. John, and L. Eeckhout. Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation. In *Proceedings of the 16th International Symposium on Computer*

Architecture and High Performance Computing (SBAC-PAD'04), pages 10–17, Foz do Iguacu, PR - Brazil, 10 2004. IEEE Computer Society Press.

[70] P. 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*, 35(2):50–58, Feb 2002.

[71] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of trace sampling for performance debugging tools. *SIGMETRICS Perform. Eval. Rev.*, 21(1):248–259, 1993.

[72] G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan. 1998.

[73] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[74] D. Noonburg and J. Shen. Theoretical modeling of superscalar processor performance. pages 52 – 62, nov. 1994.

[75] P. M. Ortego and P. Sack. Sesc: Superescalar simulator. Technical report, 2004.

[76] M. Oskin, F. T. Chong, and M. Farrens. Hls: combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 71–82, New York, NY, USA, 2000. ACM.

[77] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 76–84, New York, NY, USA, 1992. ACM.

[78] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.

[79] D. Pelleg and A. W. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734. Morgan Kaufmann Publishers Inc., 2000.

[80] D. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. August, and D. Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. *High-Performance Computer Architecture, International Symposium on*, 0:29–40, 2006.

[81] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 321–326, New York, NY, USA, 2005. ACM.

[82] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Four generations of spec cpu benchmarks: what has changed and what has not. Technical report, 2004.

[83] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 412–423, New York, NY, USA, 2007. ACM.

[84] A. Poursepanj. The powerpc performance modeling methodology. *Commun. ACM*, 37(6):47–55, 1994.

[85] D. Price. Pentium FDIV flaw-lessons learned. *Micro, IEEE*, 15(2):86 –88, Apr. 1995.

[86] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[87] R. H. M. R E Walpole. *Probability and Statistics for Engineers and Scientists*. 1993.

[88] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *In Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, 1993.

[89] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294, New York, NY, USA, 1998. ACM.

[90] T. Sherwood and B. Calder. Time varying behavior of programs. Technical report, 1999.

[91] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

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

[93] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. pages 336–347, 2003.

[94] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Trans. Comput.*, 48(11):1260–1281, 1999.

[95] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in computer architecture evaluation. *Computer*, 36(8):30–36, 2003.

[96] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. *SIGPLAN Not.*, 26(4):53–62, 1991.

[97] SPEC. Standard performance evaluation corporation. http://www.spec.org.

[98] M. Tawk, K. Ibrahim, and S. Niar. Adaptive sampling for efficient mpsoc architecture simulation. pages 186 –192, oct. 2007.

[99] M. Tawk, K. Ibrahim, and S. Niar. Parallel application sampling for accelerating mpsoc simulation. *Design Automation for Embedded Systems*, pages 1–21, 2010. 10.1007/s10617-010-9064-0.

[100] M. Tawk, K. Z. Ibrahim, and S. Niar. Multi-granularity sampling for simulating concurrent heterogeneous applications. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 217–226, New York, NY, USA, 2008. ACM.

[101] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The liberty simulation environment: A deliberate approach to high-level system modeling. *ACM Trans. Comput. Syst.*, 24(3):211–249, 2006.

[102] M. Van, T. Sherwood, B. Calder, M. V. Biesbrouck, and S. B. Calder. A co-phase matrix to guide simultaneous multithreading simulation, 2004.

[103] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2005)*, pages 47–67, Barcelona, Spain, 11 2005. Spinger Verlag.

[104] L. Van Ertvelde, F. Hellebaut, L. Eeckhout, and K. De Bosschere. Nsl-blrl: Efficient cache warmup for sampled processor simulation. In *ANSS '06: Proceedings of the 39th annual*

Symposium on Simulation, pages 168–177, Washington, DC, USA, 2006. IEEE Computer Society.

[105] F. Vandeputte, L. Eeckhout, and K. De Bosschere. A detailed study on phase predictors. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 571–581. Springer Berlin / Heidelberg, 2005.

[106] J. Veenstra and R. J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. pages 201–207, 1994.

[107] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe. Simflex: Statistical sampling of computer system simulation. *Micro, IEEE*, 26(4):18 –31, jul. 2006.

[108] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. *SIGMETRICS '05*, June 2005.

[109] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with livepoints. In *In ISPASS 06: Proceedings of the 2006 International Symposium on Performance Analysis of Systems and Software*, pages 2–12, 2006.

[110] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.

[111] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical sampling of microarchitecture simulation. In *In 20th International Parallel and Distributed Processing Symposium (IPDPS*, 2006.

[112] J. Yi, L. Eeckhout, D. Lilja, B. Calder, L. John, and J. Smith. The future of simulation: A field of dreams. *Computer*, 39(11):22 –29, nov. 2006.

[113] J. Yi and D. Lilja. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *Computers, IEEE Transactions on*, 55(3):268 – 280, mar. 2006.

[114] J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. Lilja, and L. John. Evaluating benchmark subsetting approaches. pages 93 –104, oct. 2006.

[115] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society.

[116] J. J. Yi and D. J. Lilja. Effects of processor parameter selection on simulation results. Technical report, 2002.

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

[118] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *in ISPASS 07*, 2007.