UNIVERSITY OF CALIFORNIA, SAN DIEGO

Reproducible User-Level Simulation of Multi-Threaded Workloads

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

Doctor of Philosophy

in

Computer Science

by

Cristiano Pereira

Committee in charge:

Brad Calder, Chair
Bill Lin
Harish Patil
Tajana Rosing
Curt Schurgers
Dean Tullsen

2007

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

Chair

University of California, San Diego

2007

DEDICATION

This dissertation is dedicated to my family, who have defined the person I have become. To my dad Glebson, my mom Magdalena and my brother Marcelo.

EPIGRAPH

"for Distinction Sake, a Deceiving by Words, is commonly called a Lye, and a Deceiving by Actions, Gestures, or Behavior, is called Simulation."

Robert South (1643-1716)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

None of the work presented in this dissertation would have been possible without the assertive guidance of Prof. Brad Calder, who always kept me focused and motivated about the research conducted throughout the years we worked together. Thank you for bringing me to the architecture lab, for opening the doors at Intel and for doing a great job as an advisor, whether in person or remotely.

I also would like to thank my colleagues from the architecture lab and the embedded systems lab, where I started the long PhD journey. From the embedded systems lab, I would like to thank Frederic Doucet, for the countless hours of discussions and for the camaraderie; Ravindra Jejurikar, for his stress-free way of handling things and the many discussions on various research topics; Yuvraj Aggarwal, for always trying to keep it fun. I must also thank Jeffrey Namkung and Zhen Ma for many hours spent on discussions and coffee breaks. I cannot forget my friends from UC Irvine, where everything started so many years ago. In particular, Marcio Buss, for his friendship, for the surfing together and for all the beers we drank together. From the architecture lab, at UC San Diego, I would like to thank Satish Narayanasamy, for the research discussions, for making things look simple and for the games of table tennis. I also would like to thank Erez Perelman, Jeremy Lau, Jeffrey Brown, Jack Sampson, Michael Van Biesbrouck and Ganesh Venkatesh for their help on many technical and non-technical issues.

I must not forget to thank all of the folks from VSSAD, Intel, Massachusetts, whose help and support were also indispensable for completing this dissertation. In particular, I am very thankful to Harish Patil for his mentorship, enthusiasm, perseverance and for the great advice given during my long and rewarding internship at Intel. I am also grateful to other VSSAD members: Robert

Cohn, Greg Lueck, Chi-Keung (CK) Luk, Geoff Lowney, Aamer Jaleel, Michael Adler, Mark Charney, Joel Emer and many others.

I am also grateful to Kaylene Grove, who has always supported me and given me a good reason to leave the office and go home. I am also thankful to her family, who have adopted me and made me feel at home here in the US.

Finally, I would like to thank my family back in Brazil. They are the reason I was able to make it this far. They have always supported my decisions, even when I decided to leave them to embark on a journey 6,000 miles away from them.

Chapter III contains material that appears in "Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation", Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn and Brad Calder, in *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. The dissertation author was the primary investigator and author of this paper.

Chapter IV contains material that appears in "Reproducible Simulation of Multi-Threaded Workloads for Architecture Design Exploration", Cristiano Pereira, Harish Patil and Brad Calder, submitted to the 14th International

Symposium on High-Performance Computer Architecture, Salt Lake City, UT February 16-20, 2008. The dissertation author was the primary investigator and author of this paper.

VITA

1998	Bachelor of Science in Computer Science Pontifical Catholic University of Minas Gerais, Brazil
2000	Master of Science in Computer Science Federal University of Minas Gerais, Brazil
2007	Doctor of Philosophy in Computer Science University of California, San Diego, USA

PUBLICATIONS

"Reproducible Simulation of Multi-Threaded Workloads for Architecture Design Exploration" Cristiano Pereira, Harish Patil, Brad Calder. *Submitted to the 14th International Symposium on High-Performance Computer Architecture.* February, 2008.

"Recording Shared Memory Dependencies for Application-Level Replay Debugging" Satish Narayanasamy, Cristiano Pereira, Brad Calder. *The 12th International Conference on Architectural Support for Programming Languages and Operating Systems.* October, 2006.

"Software Profiling for Deterministic Replay Debugging of User Code" Satish Narayanasamy, Cristiano Pereira and Brad Calder. *The 5th International Conference on Software Methodologies Tools and Techniques.* October, 2006.

"Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation" Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn and Brad Calder. *International Conference on Measurement and Modeling of Computer Systems.* June, 2006.

"Dynamic Phase Analysis for Cycle-Close Trace Generation" Cristiano Pereira, Jeremy Lau, Brad Calder, Rajesh Gupta. *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis.* September, 2005.

"Leakage Aware Dynamic Voltage Scaling for Real-Time Embedded Systems" Ravindra Jejurikar, Cristiano Pereira, Rajesh Gupta. *Proceedings of 41st Design Automation Conference (DAC'04) San Diego.* June, 2004.

"PASA: A Software architecture for building power aware embedded systems" Cristiano Pereira, Rajesh Gupta, Mani Srivastava. *In the proceedings of the IEEE CAS Workshop on Wireless Communications and Networking - Power efficient wireless ad hoc networks.* September, 2002.

"JADE: An Embedded Systems Specification, Code Generation and Optimization Tool" Cristiano Pereira, et. al. *Proceedings of the XIII Symposium on Integrated Circuits and System Design.* September, 2000.

"Code Generation and Optimization for Embedded Systems Specified in SDL" Cristiano Pereira, *MSc. Dissertation - UFMG (Federal University of Minas Gerais) - Computer Science Department, Belo Horizonte, Minas Gerais, Brazil.* July, 2000.

ABSTRACT OF THE DISSERTATION

Reproducible User-Level Simulation of Multi-Threaded Workloads

by

Cristiano Pereira
Doctor of Philosophy in Computer Science
University of California, San Diego, 2007
Professor Brad Calder, Chair

As the complexity of processors increases, it becomes harder for designers to understand the non-trivial and many times non-intuitive interactions among the micro-architecture internal structures. Understanding these interactions is important because it helps pinpoint bottlenecks, enabling designers to reason about sources of performance loss and improve their next generation of processors. To help designers understand these interactions in current and, more importantly, in future generation designs, designers make heavy use of computer architecture detailed simulation. These simulators model the behavior of the processor on a per-cycle basis, allowing designers to look at very detailed trade-offs. Building and maintaining these simulators is a large and complicated task. In addition, recent trends in designing micro-architectures with multiple cores in the same chip brings new challenges that affect the way simulation results should be compared. This dissertation focuses on techniques to help build and maintain simulators, as well as techniques to improve the way architects evaluate design choices using simulation.

Existing user-level simulators require manual hand coding for the emulation of each and every possible system effect (e.g., system call, interrupt, DMA

transfer) that can impact the application's execution. Developing such an emulator for a given operating system is a tedious exercise, and it can also be costly to maintain it to support newer versions of that operating system. Furthermore, porting the emulator to a completely different operating system might involve building it all together from scratch. The first contribution of this dissertation is a technique to automatically capture the system effects to an application. The system effects are captured in logs and then used to guide achitecture simulation. By using the proposed technique, the complexity of implementing and maintaining user-level simulators is greatly reduced. In addition, the technique guarantees deterministic simulation on uni-processor systems.

As multi-core processors become main stream, techniques to address efficient simulation of multi-threaded workloads are needed. Simulation of multi-threaded workloads on multi-core systems suffer from non-determinism across runs in different architecture configurations. If the execution paths between two simulation runs of the same benchmark, with the same input, are too different, the simulation results cannot be used to compare the configurations. The other contributions of this dissertation focus on techniques to efficiently collect simulation checkpoints for multi-threaded workloads. It extends the previous technique to efficiently collect logs for uni-processor simulation. Using these checkpoints, multi-threaded simulation in multi-core systems becomes deterministic. The deterministic simulation results in stalls that would not naturally occur in execution. This dissertation proposes techniques that allow one to accurately compare performance across architecture configurations in the presence of these stalls.

I

Introduction

Advances in system integration technology have enabled a steady increase in transistor density, with more transistors used by every new generation of processors. Recent processor designs, such as the Intel Dual-Core Itanium 2 Processors, released in 2006, have more than 1 billion transistors in a single chip. This abundance of transistors allows the design of complex architectures, enabling the implementation of aggressive techniques to dynamically execute instructions out-of-order, exploiting as much instruction level parallelism as possible [32]. More recently, even as the rate of performance increase achieved by exploiting instruction level parallelism bottoms out, computer architecture still continues to advance by transitioning to designs where many processing units are included in a single chip [29], known as multi-core processors.

As the complexity of processors increases, it becomes harder for designers to understand the non-trivial and many times non-intuitive interactions among the micro-architecture internal structures. Understanding these interactions is important because they determine the speed at which programs execute in that architecture. To help designers understand these interactions in current and, more importantly, in future generation designs, designers make heavy use of computer architecture simulation.

In a recent study, Yi *et al* [79] shows the trend in performance evaluation methodologies for papers accepted to the *International Symposium on Computer Architecture* (ISCA), the most important conference on computer architecture. In summary, in 1985, 7% of the papers accepted evaluated their architectural enhancements using simulators. In 2004, the number jumped to 87%. The reason for that is the complexity of recent designs. The micro-processor manufacturing industry is not different. Mainstream microprocessor design and manufacturing companies such as *Intel Corporation* and *Advanced Micro Devices* (AMD) also rely heavily on simulation to help evaluate design choices and understand the behavior of workloads running on their chips. This is also due to complexity of new designs and the prohibitive cost to build prototypes, as the fabrication processes become more expensive. Therefore, it is clear that simulators are indispensable tools for the current and future success of computer architecture research, both in academia and industry.

This dissertation focuses on techniques to help build and maintain simulators, as well as techniques to improve the way architects evaluate design choices using simulation of multi-threaded workloads in multi-core processors.

I.A How computer architects use simulation

Researchers use simulators to model the behavior of a micro-processor architecture in details. Simulators can be used for many purposes: 1) to understand the bottlenecks of the current designs; 2) to assess the viability of implementing architectural enhancements, which in many cases are proposed as a result of understanding the bottlenecks; 3) to project performance of workloads and benchmarks in new architectures; 4) to create inputs to analytical models. The *de facto* standard for computer architecture simulation models is based on execution-driven, cycle-accurate simulators. These are simulators that execute

each instruction of the workload being examined, modeling the path through which the instruction proceeds as it advances through the various stages of a processor pipeline. This behavior is modeled on a per-cycle basis, to help understand what happens during each cycle of execution, in the various processing units. They are widely used because of their accuracy in predicting performance and because of their ability to model speculative execution. They enable the evaluation of detailed trade-offs, which is not possible using other higher-level simulation models, such as analytical simulation.

To help evaluate the performance of a given micro-architecture, designers typically execute standard workloads that represent a diversity of behaviors, also known as benchmarks [3, 4, 9, 24]. When understanding the effects of a new architectural feature or enhancement, the benchmarks are run in the simulators, modeling different architecture configurations – where each configuration implements a possible variation of the enhancement (perhaps with different parameters) – in order to collect quantitative results. These results are then analyzed and a conclusion is drawn as to whether the enhancement is beneficial. Figure I.1 illustrates the scenario. A benchmark, composed of m programs, is simulated with n different configurations. Each configuration represents a design point in the space of possible choices. For example, the first configuration can be a processor with half the size of L1 data and instruction caches as configuration 2. Or configuration 1 can use a different implementation of a cache coherence protocol, compared to configuration 2. Typically one of the configurations is the baseline, which is used as a common denominator to rank the results of the other configurations [1]. The simulations for each configuration are run and the results are output. These statistics, collected during the simulation runs, are used to evaluate them. For example, if the goal of the enhancement is to maximize performance, a common

[1] Alternatively, designers may look at radically different design options. In those cases, however, higher level models are also adequate.

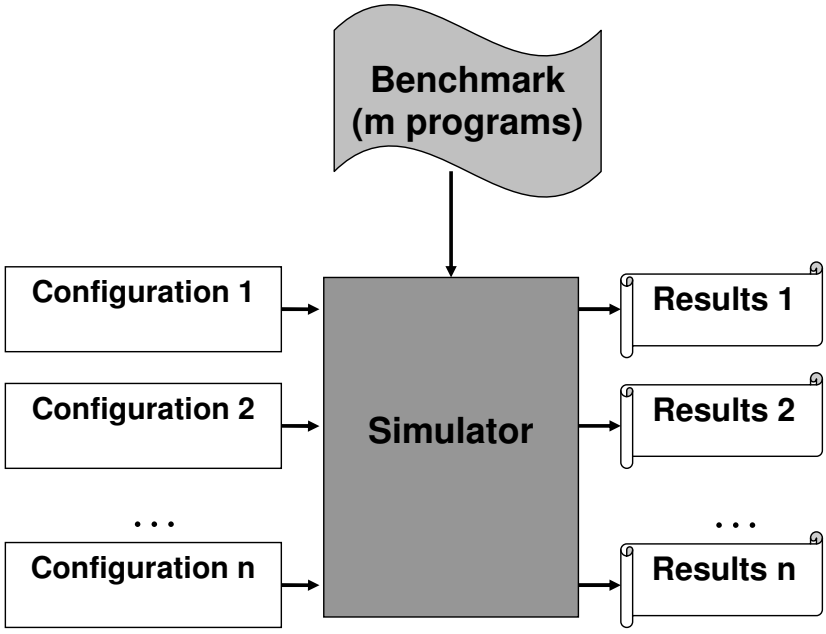

Figure I.1: Typical scenario when comparing architecture configurations. A benchmark, consisting of m programs, is simulated through n different configurations.

way to evaluate the design choices is to use a performance-related metric, such as the number of instructions completed per cycle (IPC), for a workload. Once the IPCs for the configurations are available, the next step is to compute the speed-up across the runs. If the goal is minimize energy consumption, other metrics are better suited, such as the energy-delay-product, which seeks to minimize energy without a significant impact on performance. One basic requirement for these comparisons to work is that the amount and type of work executed across the various simulation runs is the same. As will be shown later, this is not always the case, and techniques to ensure this property are beneficial.

I.B Motivation

As previously noted, a cycle-accurate and execution-driven simulator is a powerful tool to understand the behavior of a program running on a micro-architecture, and the complex interactions among internal micro-architectural structures. However, the accuracy and visibility comes at the cost of very high run-times required to complete the simulations. For instance, *Simplescalar*, a widely used simulator employed by academics, can execute less than 1 million instructions per second (on a modern Pentium 4 2.4GHz processor) when modeling a moderately complex out-of-order single core processor, with a very simple memory hierarchy. Industry simulators, on the other hand, which tend to model real architectures with very detailed and complex models, are even slower. The performance model used in Chapter IV simulates a multi-core processor, with a complex memory hierarchy and interconnection network. These models execute on the order of 1K to 10K instructions per second. This is because of the complexity of the models, the richer level of detail, and because the modular nature of the simulator's implementations, which are meant to be used for generations of processor designs. This leads the implementation of industry simulators to emphasize on re-usability and well defined interfaces across the various components modeled.

Simulators are also be classified in terms of what they model. Two broad categories are commonly found: user-level and full-system simulators. Full-system simulators simulate in detail both the user-level code, the operating system and device drivers code. They model the processor and all other peripherals needed for the correct execution of the program. These simulators can generally boot unmodified operating system codes. However, implementing and maintaining a full-system simulator is a very complex task. Even configuring benchmarks to run in those simulators can be a hard task. If the applications do not spend a

significant amount of execution in the operating system, the complexity of implementing these simulators is not warranted. Examples of full-system simulators are Simics [43], SimOS [59] and SoftSDV [70]. User-level simulators only perform detailed simulation of the user-code and the system shared libraries. Even so, the interactions with the operating system need to be emulated for the program to execute correctly during simulation. For example, if the program executes a system call to read data from a file, that system call's side effects have to be reflected in the simulation for correct execution, otherwise the program does not read the data it needs to continue its execution. Examples are such simulators *Simplescalar* [17] and SMTSim [68].

Existing application-level simulators require manual hand coding for the emulation of each and every possible system effect (e.g., system call, interrupt, DMA transfer) that can impact the application's execution. Developing such an emulator for a given operating system is a tedious exercise, and it can also be costly to maintain it to support newer versions of that operating system. Furthermore, porting the emulator to a completely different operating system might involve building it altogether from scratch. This is the first problem area where this dissertation makes contributions. A technique to automatically capture the system effects to an application in logs is proposed. A collection of logs, referred to as a *checkpoint*, is then used to guide architecture simulation. By using the proposed technique, the complexity of implementing and maintaining user-level simulators is greatly reduced. In addition, the technique guarantees deterministic simulation on uni-processor systems, which is desired for accurate comparison of configurations. Chapter III presents and discusses the proposed technique in detail.

The cost of running simulations is worsened as the length of benchmarks to be simulated increases. The latest version of SPEC benchmarks, released in

2006, executes close to one trillion instructions on average, up from an average of 114 billion instructions for SPEC2000. With the recent focus on multi-core architectures increasing attention has shifted to the simulation of multi-threaded benchmarks. These programs are good candidates to exploit the full benefit of multi-core architectures, by the use of thread-level parallelism. Examples of these programs are found in multi-threaded benchmarks such as SPECOMP2001 [9] and RMS (Recognition-Mining-Synthesis) [24]. These programs consist of multiple threads of execution, which execute cooperatively in order to accomplish a program's task. These programs also have a large dynamic instruction count, in the order of trillions of instructions.

Furthermore, multi-threaded benchmarks, when executed in multi-core architectures, present yet another challenge: *non-determinism* across simulation runs with different architecture configurations, as pointed out by Alameldeen *et al* [7] and Lepak *et al* [41]. This breaks the requirement noted earlier in section I.A, that the execution paths are executed across the simulations runs, guaranteeing that the workload is performing the same amount of work across the runs. However, this is not true for shared-memory multi-threaded programs. The non-determinism comes from the fact that threads do access shared memory locations in a different order during simulation of different architecture configurations. This is because threads' rate of progress with respect to one another changes. For example, the order in which locks are acquired by threads in one architecture configuration can be different across two runs. Also, the number of cycles and instructions spent spinning for a lock can be different. As a result, the execution paths across two executions are not guaranteed to be the same. If the variation in the execution paths is significant, two simulation runs cannot be compared directly, because the amount and type of work performed differs across executions. The problem is worsened when the operating system behav-

ior is also modeled, since changes in the architecture configuration can result in interrupts arriving at different points in the execution, causing the OS to schedule threads differently across two runs. One possible solution to overcome this problem is to increase the number of simulation runs needed to evaluate a given configuration, as proposed by Alameldeen *et al* [7] (and explained in chapter IV). This, however, increases the run-time cost of evaluating new designs significantly. This is the second problem area where this dissertation makes contributions. In particular, an extension to the checkpointing mechanism for uni-processor simulation is presented in Chapter III. This extension allows efficient capturing of enough information to guarantee deterministic execution also in multi-core architecture simulation. To guide simulation from these checkpoints, modifications in a simulator are required. These modifications make the simulation deterministic. Deterministic simulation requires the introduction of *stalls* in the simulation that would not naturally occur in the execution of the program. This dissertation proposes techniques to account for these *stalls* in order to allow designers to compare simulation runs and make decisions about them.

I.C Contributions

The complexity of building and maintaining computer architecture simulators, the need for determinism across simulations, the need for techniques for efficient creation of user-level simulation checkpoints, and the new challenges arising as a result of the multi-core era, motivated the development of the techniques presented in this dissertation. The contributions of this work are summarized as follows:

- **Automatic logging of operating system effects.** A technique and a tool to capture and log operating system effects for simulation is presented. The technique enables capturing checkpoints for user-level simulators in a very

easy and portable manner. It trivializes the need to implement emulation support in these simulators. It also enables deterministic simulation on uni-processor systems because it removes the sources of non-determinism from the simulation. This is especially important if one wants to analyze interactive applications and applications whose interactions with the outside world (e.g. network I/O) dictate its behavior.

- **Efficient capture of multi-threaded program behavior.** An extension of the tool to capture system effects enables capturing of multi-threaded program executions on multi-core[2] systems. These checkpoints can be used to guide multi-threaded workload simulation on multi-cores deterministically.

- **A technique for comparing design alternatives using deterministic simulation in the presence of artificial stalls, introduced to remove non-determinism.** The implementation of a deterministic simulator and a technique to compare simulation runs when using a deterministic simulator is presented. Deterministic simulation introduces artificial stalls to ensure same execution paths across simulation runs with different configurations. The proposed techniques show how to account and deal with these stalls.

I.D Organization

This dissertation is organized as follows. Chapter II presents a brief description of simulation in computer architecture. It describes different simulation styles and techniques to reduce and speed-up simulation. In particular, it discusses statistical and representative sampling. These are techniques used to select samples for simulation. Chapter III also explains mechanisms used to reach a sample for simulation, once it is selected. The techniques presented in

[2]The focus is on multi-core systems, but nothing prevents a user from applying the technique on multi-processor systems.

other chapters can be directly applied with both types of simulation sampling. This chapter concludes with an introduction to binary instrumentation, which is used in the implementation of the tools described in chapters III and IV. Chapter III discusses a tool, called pinSEL, and the algorithm applied to automatically capture system effects in checkpoints, which are then used to guide architecture simulation. It provides a detailed description and an evaluation of log sizes and run-time overhead to collect the logs. Chapter IV describes the extensions made to the pinSEL checkpoints to support deterministic simulation of multi-core workloads. This technique is a step towards addressing variability in multi-threaded workloads, when running in multi-processor systems. It also describes the simulation changes needed to guide simulation from these checkpoints. Chapter IV concludes with a description of the techniques to deal with the artificial stalls introduced during simulation, in order to allow design exploration in multi-core architectures. These stalls can be used to provide a speed-up estimate when comparing two designs. Finally, chapter V summarizes the dissertation and identifies future research directions.

II

Simulation Background

Simulators model the performance of a system. They are used to predict the behavior of future generation machines and to understand the performance of current machines, in order to find bottlenecks and fix performance bugs. For future generation processors, a simulator, usually implemented in software, is an inexpensive and flexible way to understand the performance, simply because the processor does not exist. An alternative would be to prototype the processor, but that is expensive, especially with today's designs, whose implementations contain more than one billion transistors in a single chip. Understanding the performance of existing processors can be done with direct measurements, but simulation models provide a much higher level of visibility, and enable the flexibility to change the sizes of internal structures very easily (e.g. cache associativity). Although inexpensive compared to building a prototype, and flexible, simulators are complex to implement and maintain, particularly as the number of cores per chip increases.

Simulators are, in general, software tools. The platform in which the simulator executes is defined as the *host*, and the architecture simulated is called the *target*. The host platform can be any architecture and does not need to be tied with the target. For instance, the host platform can be a x86 machine, and

the target a PowerPC architecture.

II.A Level of simulation detail

Simulation can be performed at various levels of detail, depending on the type of study to be done and also the amount of accuracy required. At the lowest level, a micro-architecture can be modeled at the circuit level, to verify and understand the behavior of the transistors implementing the system. Executing simulations at this level of detail is extremely slow, and not practical for a typical computer architecture study, where many millions of instructions are executed. A common practice is to use circuit-level simulators to derive analytical models, which are then used along with simulators that model the architecture at higher levels of abstraction [40]. At a higher level of abstraction there are also gate-level simulators. These simulators do not model circuits, but structures such as AND or XOR gates and their wire interconnections. Above gate-level, one can find RTL (register transfer level) simulators. These model structures such as adders and multipliers, and registers. The registers store intermediate results between computations. A common way to specify an RTL model is to use hardware description languages (e.g. VHDL). Instead of connecting gates with wires, higher level statements, such as $reg0 = reg1 + reg2$, describe the implementation of the architecture. Above the RTL level, there are detailed cycle-accurate simulators. These simulators model the behavior of micro-architectural structures and their inter-connection, on a per-cycle basis. These models are usually implemented by high-level languages such as C/C++. Typical micro-architectural structures present in these models are caches, instruction queues, re-order buffers, branch predictor tables, etc. At this level, the internal implementation of these structures is irrelevant, and only the time to perform operations and the functionality are modeled. For instance, users of cycle-accurate simulators do not care how

Table II.1: Levels of detail for architecture simulation

Level of detail	**What is modeled**
Circuit	transistor behavior
Gate	gates (AND, OR, etc) and wires
RTL	registers and arithmetic structures (ADDERs, MUXes, etc)
cycle-accurate	micro-architecture structures (branch predictors, instruction queues, caches, re-order buffers)
functional	programmer visible structures (registers and memory)
analytical	abstract structures (queues, servers, etc)

many gates are used for implementing a cache. Instead they care whether a given address is a hit or miss, as well as the latency and power to access it. Functional simulators implement yet another level of abstraction, where no internal micro-architectural structures are modeled. Rather, only programmer-visible structures exist. Finally, there are analytical models, which use theories such as queuing models or petri-nets, for instance, to model the behavior of the architecture. Table II.1 presents a summary of the descriptions above. The focus of this dissertation is on cycle-accurate simulation models, which are very popular in both academia and in industry, due to their accuracy in predicting performance. The next section expands on cycle-accurate and functional simulators, due to their importance in computer architecture research.

II.A.1 Functional simulation

Functional simulation models the functional behavior of the architecture. This involves executing or interpreting the instructions defined by the instruction set architecture (ISA) correctly. Functional simulators model the visible architectural registers and the memory states. The only goal is to correctly execute instructions from a program by updating the simulated registers and

memory. Hence, no modeling of time or internal micro-architectural structures is performed.

Functional simulation is useful for many tasks. One example is characterization of the micro-architectural independent program behavior throughout the execution. By looking at basic block profiles, instructions mixes (e.g. number of integer, floating point, memory access, control instructions, etc), one can understand the nature of the program. This is useful, for example, to create input for statistical simulators [52, 26]. Functional simulators can also emulate other peripherals, in addition to the processor. This allows software developers to write code for a future hardware platform before the platform is built.

II.A.2 Cycle-accurate (detailed) simulation

Cycle-accurate simulation not only models the functional behavior of the micro-architecture ISA, but also the timing behavior of each instruction. When executing cycle-accurate simulations, the path taken by an instruction while executing through the processor's pipeline is modeled in detail. As a result, the time to execute an instruction depends on its type (integer or floating point arithmetic, control flow, etc), and also on the state of the internal micro-architecture structures. For example, a load instruction that misses the cache will take longer to execute than one that does not. Due to the level of detail simulated, computer architecture cycle-accurate simulators are also called detailed simulators. In this dissertation these terms will be used interchangeably.

Detailed simulators are much more complex to implement, because they model the interactions between internal structures in the micro-architecture. Typically every pipeline stage is modeled. At each cycle of execution, the state of each structure is updated, and instructions advance in the pipeline according to the latencies specified by the performance model. In addition, the simula-

tors keep track of the simulated time and statistics related to each step (e.g. cache misses, branch mispredictions). Detailed simulators usually separate the functional model from the performance model. The latter is what models the timing (e.g. latencies) and functional (e.g. branch prediction outcomes) behavior of the various structures of the micro-architecture. Since hardware structures have finite sizes, resource contention also needs to be modeled accurately. For instance, if at some point all floating point functional units are busy executing instructions, a new floating point instruction ready to dispatch has to wait for a unit to be freed before it starts executing. Also, support to flush instructions in the wrong-path of execution is required, if speculation is supported. All of this detailed modeling adds to the run-time cost of executing these simulators, which is not as high as lower level simulators, but is still significant. Cycle-accurate simulators in academia execute less than one million instructions per second. In the industry, that number is on the order of tens of thousands of instructions per second.

Multi-processor cycle-accurate simulation. When simulating multiple cores of execution, the overheads are higher because work in p different processing units is modeled at every clock cycle. In addition, interactions across processors take place. For instance, a shared memory location that is written by one processor has to be invalidated in other processors' caches. This invalidation takes a given number of cycles, depending on the interconnection network. At every clock cycle the interconnection network also models the packets traveling through it. When the cycle at which the invalidation message is supposed to arrive is reached, the remote processor receives the data, causing its cache line to be invalidated. If a multi-processor simulator is implemented sequentially, there is p times as much work to do, where p is the number of processing units in the target, in addition to the work to model the inter-connection network.

II.A.3 Detailed simulation methodologies

There are traditionally two styles of detailed simulation: *trace-driven* and *execution-driven*.

Trace-driven

In trace-driven simulation, as the name implies, a trace of events is fed to the simulator. For a cache simulator, an event can be a memory access event, represented by an effective address and the type of access (read or write). For a branch simulator, a trace can be a sequence of program counter values and a flag, indicating if the branch is taken or not. Traditionally, trace-driven simulation consists of three stages: trace collection, trace reduction and trace processing. The events can be collected in different ways. A common way is to use hardware support for collecting them by probing the system buses. Another way is using binary instrumentation [65, 42]. Binary instrumentation allows the registration of call-backs, which are executed during run-time. The call-backs can specify arguments through which the architectural state is passed, which is then output as traces. Trace compression can be implemented by using a standard compression algorithm (e.g. Lempel-Ziv), trace filtering (storing partial addresses, for cache lines only) or trace sampling. Uhlig and Mudge [71] present a comprehensive survey on trace-driven memory simulation, covering techniques for collection, compression and processing.

Trace-driven simulation feeds a fixed trace of events, regardless of the feedback from the timing model. This fixed trace is the sequence of events observed when the trace was collected. Typically, only the committed path of execution is captured. As a result, speculative execution cannot be modeled. One could obviously augment the trace with alternative paths for speculation, but that adds complexity and increases the sizes of the traces. Another issue

with trace-driven simulation is that it is hard to accurately model the behavior of modern processors, which execute instructions out-of-order dynamically and very aggressively. For example, for a trace of memory accesses, it is hard to model the correct latencies between memory operations accessing the cache, and in what order. This is because this information depends on aspects other than the address and the memory operation type. These other aspects are the latencies for executing other instructions, on which the memory access instruction depends, the type of the instructions, and internal structures of the pipeline. To model these correctly, more information needs to be added to the trace, which also adds complexity and increases the trace size.

In spite of the drawbacks previously described, trace-driven simulation does have its advantages. Trace-driven simulators are easy to implement, because they do not need to model the functional behavior of the architecture, but only interpret the events in the trace. The simulations are also completely reproducible, a consequence of the inflexibility inherent to the traces.

Multi-processor trace-driven simulation. Trace-driven simulation has also been used to study performance of multi-processors. On multi-processors, the interleaving of the events depend on the latencies associated with the target micro-architecture. A trace represents one possible interleaving, which occurred when it was collected. When simulating different configurations though, the interleaving can change. The changes in interleaving can result in different execution paths across the different runs. Un-protected access to data (data races) and synchronization operations are examples of such occurrences. When using traces, however, even if the interleavings change, the change does not affect the traces. This can result in inaccuracies because the traces from the various processing units deviate from the original parallel behavior, and the new parallel behavior traced is not coherent with the interleaving, because the trace is fixed. The phe-

nomena has been defined by Dubois *et al* [25] as *trace-shifting*. It can result in incorrect logical behavior and timing predictions. Koldinger *et al* [35] showed that different runs in multi-processors generate different traces, and these traces result in a different number of cache misses for different cache block sizes. This is because of the variability across different runs for collecting the traces. This variability has been recently defined by Alameldeen *et al* [7] as *space-variability*. Space-variability comes from the fact that different interleavings for shared memory updates are resultant in each run. The different interleavings happen because the relative progress of threads differs from run to run when running on real hardware. This is due to differences in the environment across the runs, different OS scheduling, levels of bus congestion, arrival time for interrupts or differences in the system load when running a program on a real machine. Koldinger *et al* [35] showed that, for comparing simulation results across two cache block sizes, one has to either average out the results from the different traces, for each configuration, or compare the simulations of the same trace. They observed that when the same trace is used across configurations, the trend in cache misses is the same for all the traces. However, it must be pointed out that errors due to trace-shifting are not accounted for, and those errors can affect the results of the traced run. A technique is presented in chapter IV to account for these variations by stalling threads when the behaviors deviate, providing an error metric for the performance estimates. In addition, the technique uses execution-driven simulation, with some restrictions to guarantee determinism.

Execution-driven

Execution-driven simulation [1], on the other hand, does not rely on a fixed trace of events. Instead, it fetches the actual instructions from the program

[1]Instruction-driven was the common term used in the 80s and early 90s; execution-driven at those times referred to what is now called direct-execution.

binary, decodes and executes them just like the real hardware would. On a branch prediction, the predicted program counter determines the next instructions to simulate. Since the actual instructions have all the information necessary to simulate its path as it advances in the pipeline, a very accurate simulation model can be built and executed. On a detailed simulator, instructions are only executed when their operands are available, and there are free hardware structures to accommodate them. Execution-driven simulation also allows accurate modeling of speculation. When a branch is mispredicted, the simulator executes the wrong-path instructions until it finds out that the path is incorrectly predicted. At that point it kills all the instructions in the wrong-path and flushes them out of the pipeline, rolling back any effect they may have enacted. The dynamic behavior of the micro-architecture is therefore modeled with much more accuracy than a trace-driven simulator. This accuracy, on the other hand, is also a disadvantage of execution-driven simulation, because of the complexity necessary to implement it.

Multi-processor execution-driven simulation. These simulators model multiple processors or cores of execution. At every clock cycle, each processor advances its own instructions in the pipeline. On top of the additional slowdowns, multi-processor simulation also suffers from non-determinism when executing multi-threaded cooperative workloads. Alameldeen *et al* [7] point out that multi-threaded workload runs on multi-processor simulators are non-deterministic across different architectural configurations. This is referred to as *space-variability*. As a result, two simulation runs cannot be compared directly because they are not guaranteed to execute the same paths, as in most single-threaded workloads. Consequently, one does not know if the differences in performance are because of the architectural change or because different paths were executed. In single-threaded workloads, the sequence of committed instructions

is usually deterministic.[2] As a result, comparing multi-threaded workload simulation runs in multi-processors requires a different methodology. Alameldeen *et al* [7] showed that for a fair comparison, each workload needs to be run for n times for the same configuration to obtain average behavior in each one of them. However, for the same configuration, simulators are deterministic, differently from real machine runs as shown by Koldinger *et al* [35]. To obtain variability during simulation for the same configuration, they proposed the insertion of random perturbations in the latencies to access memory. This results in variable behavior even when running the same configuration during simulation. Once a program is run for n times in each configuration, the average results can be used to make decisions. Using statistical techniques, they can choose n to give a certain confidence on the results. For very small architectural configuration changes, n can be quite large, as shown by Lepak *et al* [41]. The cost of running n times is increased run-time for performing simulations, which is now n times more expensive when using multi-threaded workloads. Given simulators' speed and length of benchmarks, the run-time cost increases quite significantly, especially when many configurations are to be explored. Barr [11] also demonstrated the variability across runs of multi-threaded benchmarks. Lepak *et al* [41] proposed a technique for deterministic-simulation, where a single run of a benchmark in each configuration is used to compare the runs. Their technique uses logs to guide the simulator and ensure determinism by introducing artificial stalls. A determinism-stall error metric is used to estimate the error in simulation results. The simulator then ensures the same path of execution for each thread. The technique proposed in Chapter IV builds upon the same idea. However, the method to collect the logs is more efficient and can be used with large applications. The mechanism to account for the artificial stalls and compute the performance estimates is also

[2]There can be examples where this is not true. Some system calls are inherently non-deterministic. Even some libraries can change their behavior depending on the architecture configuration.

improved significantly.

II.B Full-system and user-level simulation

Some benchmarks spend significant time executing operating system code. Example are I/O bound applications like TPC-C [22] and other server applications like DSS (Darwin Streaming Server) or web servers. For accurate performance modeling of these applications, it is important to execute not only the application code, but also the operating system code (e.g. system calls, interrupt handlers, etc). Full-system simulators perform detailed simulation of the user-code, shared libraries, operating system code and device-drivers as well. Hardware structures such as I/O devices, DMAs, interconnection buses, network devices, and timers need to be modeled in these simulators, so the operating system code and device-drivers execute correctly. Although very detailed, maintaining full-system simulators is a complex task. These simulators need to be capable of booting and executing un-modified operating system and device drivers code. Supporting different versions of operating systems requires constant updating of the simulators. Another obstacle for full-system simulation is the difficulty to reproduce the complex environments needed for by real applications. These applications may require special run-time license-checking mechanisms, special device drivers, they may have specific kernel dependencies, large storage requirements and elaborate installation procedures, all of which are non-trivial tasks to accomplish. Therefore, getting latest versions of some applications to run in those simulators is a time-consuming task. Hence, the complexity of such simulators is warranted only if the applications spend a significant amount of time in the OS code.

User-level simulation performs detailed simulation of the user-level (application) code and the system shared libraries only. For correctly executing

programs, these simulators need to emulate the behavior of the operating system, and reflect the side effects of the system interactions in the application's execution. These side effects result from system calls, asynchronous interrupts and DMA transfers. The emulation is a complex task, but it is significantly simpler than simulating the behavior of the full machine. Typical user-level simulators, however, only emulate a limited set of system interactions, in particular system-call interactions. This is due to the tedious and time-consuming nature of the emulation. The traditional techniques for emulation are also hard to port across different operating systems because they rely on the host platform for the emulation, tying the simulator to it. Chapter III explains in detail how the traditional emulation techniques work. Furthermore, asynchronous interrupts and DMA transfers are not emulated, limiting the scope of possible applications that can be simulated. If the benchmarks to be studied do not spend significant time in the OS code, using user-level simulators is a better option because they are simpler to build, modify and maintain.

Chapter III proposes a scheme to collect logs for user-level simulation trivializing the need for emulation support. This scheme relies on a binary instrumentation tool we created. The log collection mechanism is simple, automatic and independent of the OS. Furthermore, the problems with reproducing the execution environment to run an application are minimized. If the application can run on the native environment, it can be analyzed by our tool, which creates logs to guide simulation.

II.C Reducing the amount of simulation through sampling

Given today's simulator speeds and the dynamic instruction counts of benchmarks, it is impossible for designers to simulate full program runs in order

to evaluate a new architectural enhancement. Full runs of SPEC2000 benchmarks would take, on average, more than a month to complete. On more detailed models, such as the ones used in industry, full runs of SPEC2000 benchmarks would take years. With multi-processor architecture and multi-threaded workloads, non-determinism makes the problem even worse. As a result, a lot of research has focused on techniques to reduce the amount of simulation. In particular, many have focused on techniques to simulate only selected samples of execution for a given benchmark, because of their dramatic impact on the run-time cost of running the simulations. A sample, in this context, is an interval of execution.[3] An interval of execution is a contiguous sequence of dynamic instructions executed by a program. The goal of sampling is to simulate enough intervals of execution that, together, capture the time-varying behavior of a program. Sampling is used to estimate some program characteristics such as the average number of instructions per cycle (IPC). The number of samples and their length determines the accuracy of the sampling mechanism. If the program does not present much time-varying behavior throughout the execution, a small number of samples should capture its behavior. If the program presents a lot of variation in its behavior over time, more samples are needed to capture them all. In order to simulate the samples of execution for a given set of programs, techniques to choose and simulate those samples are needed. This section focuses on these two aspects and first discusses how to simulate a sample. There are two methods for doing this. The first executes simulation in a fast mode, called *fast-forwarding*, up until the start of the sample. At that point, the simulation switches to a detailed simulation mode and starts collecting statistics. The second collects checkpoints of the program execution at the beginning of the samples, and later loads them into the simulator. For both methods, the architectural state at the beginning

[3]Computer architects often refer to a sample as an interval of execution. A set of samples from a program is used to estimate statistics from that program. In statistics, however, a sample usually refers to a collection of sample units or measurements.

of the sample is not the same as it would be, had the program executed detailed simulation from the beginning. This results in bias for the simulation results. This section also discusses techniques to minimize the bias.

II.C.1 Reaching the simulation samples

Fast-forwarding

In general, for maintainability and modularity reasons, simulators separate functional simulation from timing simulation (or detailed simulation). This separation allows a simulator to execute programs in these two different modes: one performing only functional simulation and another performing both functional simulation and detailed simulation. The former is commonly called *fast-forwarding*, because the execution of the program advances forward at faster speeds than using detailed simulation (since no timing is modeled). When switching from functional model to detailed mode, the programmer visible state is carried over to the detailed simulation, which then starts collecting statistics. When switching from detailed simulation to functional simulation, timing modeling and statistics collection stops, and functional simulation only resumes. Fast-forwarding is commonly used to advance to samples of execution.

Cold-start effects. When switching from functional to detailed simulation, many of the processor structures modeled are in a cold state. This is because no instructions were using them in functional mode. Examples are caches, branch predictor tables and coherence directories, which are empty. Hence, some action needs to be taken to bring the simulator into a state close to what it would have been, had detailed simulation executed from the beginning and up to the start of the sample. This is referred to as *sample warm-up*. When using fast-forward, one simple way to do so is to implement yet another simulation mode, called *functional warming*. In this mode, functional simulation is executed, but

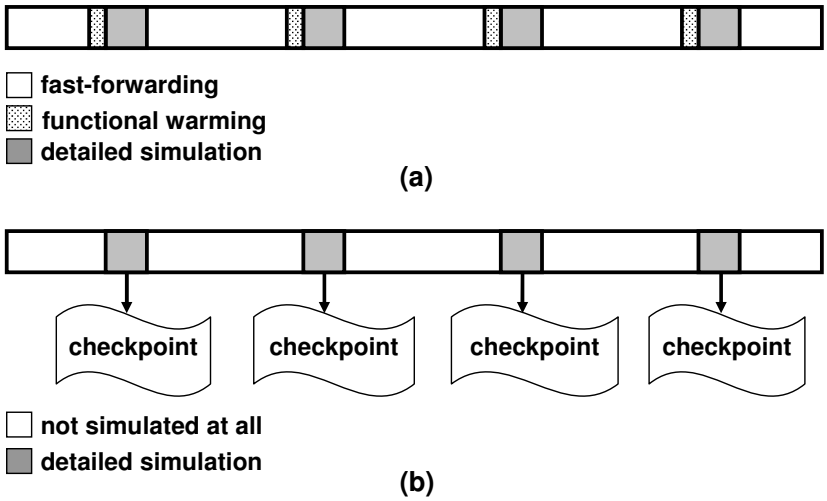

Figure II.1: (a) On-line sampling; (b) off-line sampling, using checkpoints.

major processor structures such as cache and branch predictors are still accessed. This ensures that these structures are not empty.

The amount of functional warming required before a sample depends on the sample size (number of instructions) and the warm-up accuracy desired (relative to a perfect warm-up approach, which is the state the micro-architecture structures would be if detailed simulation was executed from the start of the program). Haskins and Skadron [31] use an approach called *memory reference reuse latency* (MRRL). MRRL analyzes the program's code to determine the number of instructions to be used for functional warming. This number guarantees that a given percentage of accesses (e.g. 99%) in the sample are warmed up. Another simpler but less accurate technique, is to use *trace-stitching* [6], originally proposed with trace-driven simulation, but also applicable to execution-driven simulation with fast-forwarding. In trace-stitching, the final state of the previous sample is used to approximate the state of the next sample. Trace-stitching only performs well if the number of cache misses that would have occurred, between the samples, is confined to a small set of cache lines. Hence its performance de-

pends on the applications, sample size and the sampling period. Figure II.1-(a) illustrates a simulation using fast-forwarding, functional warming and detailed simulation. A program starts execution from the beginning in functional mode. Before a sample starts, execution is switched into functional warming. Once the sample is reached, detailed simulation takes place. Using SimpleScalar [76], functional simulation was reported to be 60x faster than detailed mode. Functional warming, which updates only cache and branch prediction structure, is 75% slower than functional simulation only. The disadvantage of using fast-forwarding is that the execution of each sample is serialized. In addition, it can still take a long time to fast-forward a program and simulate all of the samples, even in functional mode. Techniques have been proposed to speed up fast-forwarding using JIT compiling [42]. Even so, fast-forwarding time is not completely removed, and for large applications, which execute trillions of instructions, it can still take significant time. To address that, checkpointing is commonly used.

Checkpoints

A *checkpoint* can be defined as a snapshot of the program's execution, which is stored to a file, and later loaded into the simulator. Collecting the checkpoints involves using fast-forwarding to reach the samples. At that point, a checkpoint for the sample is recorded. In one fast-forwarding run, a checkpoint for each sample can be stored. In later runs, fast-forwarding is no longer needed, since each checkpoint can be loaded directly into the simulator. Using checkpoints eliminates the fast-forwarding time from the simulations. In addition, it enables parallel simulations of all samples, because the samples are independent. Figure II.1-(b) illustrates the use of checkpoints, assuming that the checkpoints were already created. During simulation, no functional execution is performed, but the checkpoints are used to simulate each sample. A checkpoint contains two

types of information: programmer visible state and micro-architecture state.

Programmer visible state. The programmer visible state consists of the register states and the initial memory values needed by the program. For user-level simulators, the checkpoints also need to contain the effects to the user-code coming from system interactions (through system calls, DMAs and asynchronous interrupts). For the memory image in the checkpoint, one can naively copy the entire memory image to the checkpoint. However, during the simulation of a sample, only a subset of the memory image is actually used by the program. In this subset, some locations in the image are read and some are written to. Only the locations which are read for the first time during the execution of the sample, before being written to, need to be in the checkpointed memory image. This is because the locations which are written to first will have the correct values in memory. These values are generated by the write operations. A similar observation has been made in other related work [14, 50]. Chapter III presents a technique to efficiently and automatically collect the system side effects and the memory state. Full-system simulators need to store more state in the checkpoints in order to start the simulation at a consistent state. In addition to the memory used by all applications currently running, it is often the case that the disk image also need to be stored, so that I/O operations executed during simulation of the checkpoint are properly handled. The memory state of all running processes also goes to the checkpoint. As a result, these checkpoints can be quite large. Furthermore, the checkpoints are not proportional to the size of the samples. Using schemes to reduce the amount of simulation through sampling can require a large number of checkpoints, in which case reduced checkpoint sizes is desirable.

Micro-architecture state. The programmer visible state has enough information to correctly start the functional simulation of the program. However, the state of the internal micro-architecture structures (e.g. caches, branch pre-

diction tables) is cold, similarly to fast-forwarding. This is because no simulation has executed before the checkpoint is loaded. As a result, a checkpoint may also carry information to warm up the architectural state. Cache and branch predictor states are the most important structures to warm up in uniprocessors. When a structure is not warmed-up, the simulation results are affected. For example, in the case of a branch prediction, more mispredictions are potentially found during the simulation of the sample because the state of the branch predictor table was cold. This difference in simulation results is called a *bias*. The simplest strategy is *no warm-up*, where no information to initialize internal architectural state is provided. For this strategy, long samples are the only way to minimize the bias in simulation results [53]. Other warm-up strategies also exist. For the memory hierarchy, *hit on cold* is one strategy, where every first cache access is considered a hit. This technique works well if the program's hit ratio during the sample is high. Fixed length warm-up is another strategy, in which one uses a period of detailed simulation prior to the beginning of a sample [20]. The MRRL technique presented in the previous section, by Haskins and Skadron [31], can also be used to create the warm-up portion of a checkpoint. Creating a checkpoint to warm up a micro-architectural structure is beneficial because it avoids using functional warm-up. However, if the information in the checkpoint is specific to a given structure configuration, the warm-up information is only useful for that configuration. If this is the case, different checkpoints for a sample are needed, for each configuration to be studied using simulation. Hence, a checkpointing mechanism that stores micro-architectural information, independent of the architecture configuration, is desired. Van Biesbrouck et.al. [14] proposed an approach called *Memory Hierarchy State* (MHS), which stores cache warm-up information for the largest cache to be explored. The MHS is collected through functional simulation. The warm-up information can then be re-used to warm-up smaller

caches as well, making the structure independent of the micro-architecture.

Checkpointing for multi-processor simulation

Checkpointing is also applied for simulation of multi-processor platforms. In this context, a programmer visible state is needed for each processor or core. In addition, warm-up state is needed for structures other than caches and branch predictors. Barr [11] proposed a scheme to warm up both the caches and the cache coherence directory, assuming a directory-based protocol. This scheme is called memory timestamp-record (MTR). MTR stores cache information and timestamps for the last readers and the last writer to each cache block. Each cache block carries a timestamp, a tag, and bits indicating whether the block is dirty and/or valid. The k most recent cache blocks to a cache set are stored. This allows the warm-up of different cache sizes and associativities, assuming a least-recently-used replacement policy. These checkpoints are also micro-architecture independent, which is a desired feature, as noted previously. The timestamps allow the re-construction of the directory, because one can infer the interleavings for cache accesses using the timestamps. Wenisch *et al* [74] use Simics [43] full-system checkpoints for saving the programmer visible state and the cache state for simulation. They use detailed simulation to warm up other micro-architectural structures such as micro-processor interconnect queue states.

II.C.2 Choosing the simulation samples

In the previous section, the techniques to avoid detailed-simulation of an entire program were discussed. However, the criteria for selecting a set of samples for simulation was not addressed. This section describes the most common methods. There are two common sampling techniques used in computer architecture simulation studies: *statistical sampling* and *representative sampling*.

Both require simulation of only a small fraction of the execution (less than 1%) with low error rates (within 3%) [80, 33], when compared to the full program detailed simulation.

Other techniques were commonly practiced until recently, using a single sample, selected by some ad-hoc criteria. One method is to perform detailed simulation only on the first X instructions or to fast-forward Y instructions and perform detailed simulation on X instructions. These techniques have been shown to poorly represent the entire behavior of the program. Other techniques have also been proposed, such as reducing the input set for running a given workload. The basic idea is to modify the original reference input so that the benchmarks run to completion, but with a much lower instruction count than the original reference. MinneSpec [34] is one example of this technique. Unfortunately, Yi *et al* [80] has shown that the technique performs poorly when compared to the reference run results. This is because running a reduced input set does not exercise the simulation in the same way that the reference input does.

Figure II.2 represents the execution of a program as horizontal bars, where shaded areas represent detailed simulation. Figure II.2-(a) illustrates full-program execution using detailed simulation, which is impractical due to the high run-time cost. The next two sections describe statistical and representative sampling.

Statistical sampling

Statistical sampling [76, 21, 39] is based on well-known theory for estimating a property of a population using samples, or a subset of the population, which are taken at random. By looking only at samples, one can infer the property for the entire population. The estimated property can be the mean of the population, its variance, or some other characteristic. In order to give the de-

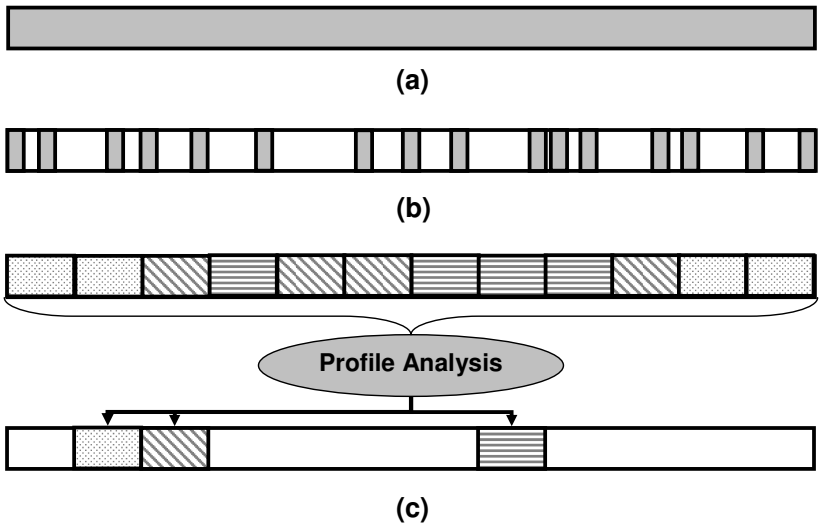

Figure II.2: (a) Full program detailed-simulation; (b) Statistical sampling; (c) Representative sampling.

signer some certainty about the estimated property, it is common practice to compute the statistical confidence of the estimate, using a confidence interval, along with a confidence level.

SMARTS is a framework to perform statistical sampling for micro-architecture simulation. In SMARTS [76], the user chooses a desired confidence and an initial number of samples n. Detailed simulation is only performed in each sample. A characteristic of the program, such as its IPC, is then estimated from these samples, and a confidence on the estimate is calculated. The confidence is quantified by a confidence level $(1 - \alpha)$ and a confidence interval $\overline{x} \pm \varepsilon.\overline{x}$, where $\overline{x}$ is the population mean estimate and ε is the error in the estimate. The confidence interval, with a confidence level α, for a population's mean μ is computed as $\overline{x} \pm \varepsilon.\overline{x}$ where ε is given by $\frac{(z.\overline{V_x})}{\sqrt{n}}$, z is the $100[1 - (\alpha/2)]$ percentile of the standard normal distribution, $\overline{V_x}$ is the estimated coefficient of variation, and n is the number of samples. The interpretation of the confidence interval, in this

case, is that α percent of all possible samples, from that population, result in an interval that captures the true mean μ [76, 23]. These formulas work for a number of samples sufficiently large (i.e. $n > 30$).

Given an initial number of samples n for a program, and a coefficient of variance $\overline{V_x}$, one can compute the confidence interval for a given confidence level or vice-versa. If the confidence is not acceptable, a new n is chosen, and the experiments are re-run. Figure II.2-(b) illustrates the statistical sampling approach. The shaded areas represent the program intervals where detailed simulation is executed. The areas in white are the portions of the program not simulated in detail. Wunderlich *et al* [76] showed that statistical sampling results in error rates of $\pm 3\%$ with a confidence level of 99.7%. It results in speed-ups of 35x to 60x when compared to full-detailed runs (shown in Figure II.2-(b)). The drawbacks of statistical sampling are that a large number of small samples need to be taken throughout the entire execution of the benchmarks. If fast-forwarding with functional warming is used, this can take a significant time, dominating the simulation time. If checkpoints are used, a large number of checkpoints are needed, which can be inconvenient and result in large storage requirements. Additionally, since the intervals representing a sample unit are very short (e.g 10,000 instructions or so), warm-up of micro-architecture structures must be done carefully. To mitigate these issues, Wenisch *et al* [73] proposed the use of statistical sampling with checkpoints, where the checkpoints are called *live-points*. These contain only the live-state needed by the sample, similar to the work presented by Van Biesbrouck [14].

Statistical sampling for multi-processors. Statistical simulation has been largely applied for single-processor architecture studies. Only recently, efforts have been directed to multi-processor simulation. Ekman and Stenstrom [27] showed that the number of samples needed to capture the time-varying behav-

ior of multi-threaded programs is usually lower than single-threaded programs. They show that a factor of p reduction can be obtained, where p is the number of processors. This is true when the various threads of execution are not performing the same work at the same time. In other words, their behavior is not aligned (e.g. through the use of barriers, for instance). As a result, the overall variation, when looking at the aggregated results (e.g. IPC) of all threads, is smoothed out, requiring less samples. This is useful when the designer is only interested in the overall behavior and not the behavior of each thread. When the latter is true, the number of samples does not decrease. However, if the threads are highly synchronized and perform the same work at the same time, the variation does not decrease, nor does the number of samples. It should be noted that none of these approaches consider space-variability in the simulation samples when running the simulations across different configurations.

Representative sampling

Representative sampling [36, 55, 62] does not rely on samples taken at random. Instead, it relies on profiling an architectural independent property of the program's execution, and using the profile to intelligently choose the samples. Program structures used to build the profile can be basic block profiles [61], loop branches, instruction mix, memory address information, register usage and procedures [38]. A program is broken into intervals of execution and a profile is collected for each interval. These profiles can then be analyzed using machine learning techniques, such as clustering, to find out a set of intervals that represents the execution of the entire program, thus capturing its time varying behavior. Only those intervals are simulated in detail.

SimPoint. The SimPoint [62] representative sampling approach picks a small number of samples, which accurately create a representation of the com-

plete execution of the program. It breaks a program's execution into intervals, and for each interval it creates a code signature (or profiles based on the code executed). It then clusters intervals with similar code signatures into phases. The idea is that intervals of execution with similar code signatures have similar architectural behavior, and this has been shown to be the case by extensive research [62, 37, 53, 80]. Therefore, only one interval from each phase needs to be simulated in order to recreate a complete picture of the program's execution. SimPoint then chooses a representative from each phase and performs detailed simulation on that interval. Taken together, this sample of intervals can represent the complete execution of a program. The set of chosen sample intervals is called *simulation points*, and each simulation point is an interval on the order of millions of instructions. Each simulation point is run through detailed simulation, allowing one to project the performance of the program based on the results from the sample. Since the intervals of execution used by SimPoint are large (e.g. 100 million instructions), issues related to warming up micro-processor structures are minimized. In addition, since the number of simulation points for representing the full execution accurately is small (e.g. 10 for SPEC2000), the number of checkpoints needed is also small. SimPoint accuracy is within 2%, relative to the full detailed simulation runs of SPEC2000 programs.

Figure II.2-(c) illustrates how representative sampling works. The program is broken into intervals and the profiles for each interval differ from one another, represented in the figure by different patterns. However, some intervals are similar to others. The key idea is that intervals with similar profiles (same patterns) are repetitive behaviors recurring over time. These behaviors do not need to be simulated over and over again. The profiling analysis discovers this and picks one sample from each group of intervals with similar behavior.

Representative sampling for multi-processors. Representative

sampling has been used in industry [53] and in academia for running architectural simulations for single-threaded programs on uni-processors. Representative sampling, in particular SimPoint, has also been used by Van Biesbrouck *et al* [15] for detecting phases in multi-programmed workloads on SMT processors with more than one context. In their work, phases for each program are identified individually. After that, a co-phase matrix is built, for the possible combinations of phases across the various programs. This co-phase matrix is used to guide detailed simulation of the phase combinations, and gather the statistics for each of them. It is also used to guide fast-forwarding when a phase combination already simulated is reached again. Results from their research show that only 1% of a workload is simulated in detailed mode, with accuracy within 4%, when compared to the full-detailed simulation runs of the workload. In multi-programmed workloads, the behavior of each individual program does not change as the architecture configuration is modified because there is no sharing of memory across the programs. Hence, there is no space-variability to deal with.

For multi-threaded programs that share memory, a complete solution for selecting representative samples, and simulating them for design exploration has not been provided yet. Perelman *et al* [56] proposed a methodology for selecting simulation points for multi-threaded programs. In their work, a profile for each interval of execution per thread of the program is created. These profiles are fed to SimPoint together, which then finds the phases and selects samples. Figure II.3-(a) illustrates SimPoint for single-threaded programs. Figure II.3-(b) illustrates the process for multi-threaded programs. A profile for each thread (e.g. $T1, T2, ..., Tn$) is collected. The figure shows each thread with a different shade. These profiles across all threads are fed to SimPoint, which selects the simulation samples from them altogether. The simulation samples are selected on a per-thread basis. This is in contrast with a possible approach in which

profiles could be combined across all threads, representing the parallel behavior across them. After the phase analysis is finished, each interval from each thread is assigned to a phase. In order to get a global picture of phase behavior across all threads, the intervals of execution from each thread must be aligned according to what was observed during profiling (the profiles in Figure II.3-(b) are unaligned specifically to illustrate this phenomena). A simulation sample across all threads is the simulation interval of execution picked by SimPoint from one thread, along with the intervals that were executed in parallel in the other threads. Perelman *et al* [56] found that the phases across threads align quite well. This is because, in these applications, each thread is executing the same code, and also because the execution of the threads is very synchronized. Is it also worth mentioning that the simulation points are selected based on one profile from one run. However, when doing design space exploration across configurations, the profiles can change due to space-variability. As a result, the profile of the code simulated in a different architecture can be different from the profile selected by SimPoint. Therefore, comparing the simulation across different architectures can lead to inaccurate results, because execution paths differ across the runs. The technique presented by Alameldeen *et al* [7] can be used in this case, with the added cost of running simulations multiple times. Even so, if the profile of the simulated run changes, the sample may no longer be representative. Alternatively, the techniques presented in Chapter IV, used for deterministic simulation for design space exploration, can be directly integrated with such an approach for representative sample selection. More discussion on this is presented chapters IV and V.

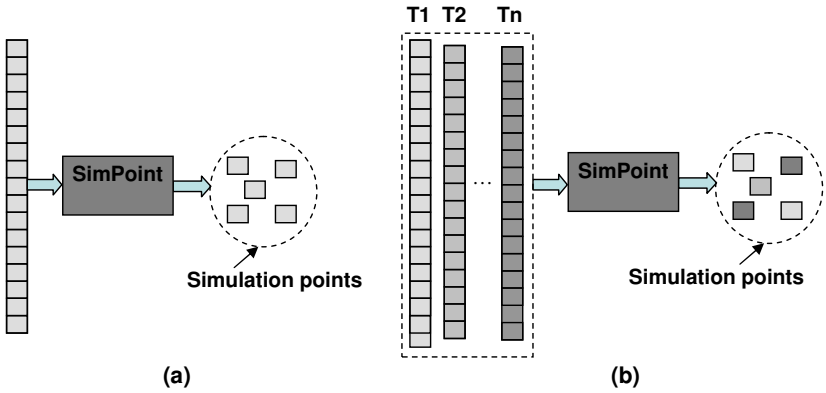

Figure II.3: (a) SimPoint for single-threaded program; (b) SimPoint for shared-memory multi-threaded programs.

II.D Accelerating simulation

The techniques presented in the previous section reduce the amount of simulation by means of simulation sampling. There are also techniques to accelerate the simulation of the samples. The techniques described in this section strive to reduce the overall time to execute all the samples of execution from a benchmark, for the different configurations examined.

II.D.1 Using parallel hosts

One obvious and easy way to accelerate the time to run simulations is to take advantage of a batch system, in which several nodes of computation execute jobs in parallel. The goal of the batch system is to schedule, prioritize and load balance the execution of the jobs. As described in Chapter I, designers evaluate n variations of an architectural enhancement by simulating m programs. There are therefore $n.m$ independent tasks that can run in parallel. In addition, the individual experiments can also be run in parallel through the use of checkpointed samples. Assuming k samples per program, there are now $n.m.k$ jobs that can be run in parallel. Although this can provide significant reduction in time to execute

the simulation runs, the end-to-end time to run each job still plays a key role in the turn-around time of experiments. Running experiments is an interactive task and it is often the case that the completion of previous experiments determines the new set of experiments. Hence, the quicker a set of experiments is finished, the quicker decisions are made, and consequently time-to-market for a product decreases.

In order to fully exploit the availability of parallel hosts, one can also parallelize each simulation run, so that the simulation consists of many threads of execution, each running on a separate processor. Multi-processor simulation models a target that is intrinsically parallel. One intuitive way to break up the simulation in threads of execution is to map each target processor to a host processor. This allows the simulation to take advantage of the parallelism available in the host platform. However, a major challenge lies in ensuring proper synchronization for the host parallel computations, so that the target processors' interactions are modeled correctly.

The Wisconsin Wind Tunnel [48] is an example of a simulator that takes this approach. As long as the target processors do not interact, the threads of execution can execute in parallel, in each target processor. However, when there are interactions, threads need to be synchronized correctly. For example, in shared memory architectures, a write to a shared variable must eventually be visible to other processors. Assume that a write at cycle t must be visible by another processor at cycle $t + \delta$, according to the model and the state of the interconnection network. The other processor execution must be able to detect that event before its simulated time exceeds $t + \delta$. One simple way to implement this is to synchronize all the target processors at every target simulated cycle. A well-known strategy to synchronize computations in parallel multi-processor is to break the target processors' execution into lock-step intervals named *quanta*. All

the host processors must synchronize at every quanta, to exchange events and properly model the target processors interactions. The granularity of the quanta determines the accuracy with which the interactions are modeled, because the communications across processors are only noticed at the end of the quantum.

The Asim simulator [28] was also modified to incorporate parallel processing. Asim is a very modular simulator, in which each hardware structure is implemented by a module. Modules communicate through ports, which enforce latencies when modules communicate. By adhering strictly to these ports, for communication, the modules (in this case, processors) can be executed by threads, enabling each target processor to be executed in a host processor [10]. Mermaid is another example of an implementation of a parallel simulator for parallel model [57]. They separate computational models from communications models. The communication model is implemented by a single thread and communicate with the computational models through message passing.

II.D.2 Direct-execution

In execution-driven simulators, every instruction of the target architecture is interpreted by the simulator. This allows the simulator to model the behavior of the instructions in any level of detail necessary. The cost involved is proportional to the amount of detail, and to the fact that instructions are interpreted. Direct-execution executes the target instructions directly in the host, natively [75, 47]. The gains in speed come from the fact that blocks of instructions are executed natively, and the simulator is only invoked at certain events, such as branches, or when simulated processors need to interact. Direct-execution could invoke the simulator at each instruction boundary, but the gains in speed would be reduced. The Wisconsin Wind Tunnel [48] makes use of direct-execution to improve speeds of memory simulation. For simulation of a memory hierarchy, for

example, they instrument every load and store to invoke the simulator and handle the simulation of the memory events. Since direct-execution does not invoke the simulator frequently, in order to maximize speed, the accuracy of the simulation is usually reduced. Direct-execution also ties the host and the target platforms, since the same instruction set is required for it to work.

II.D.3 FPGA-based simulation

The simulators referred to thus far are all implemented using high level languages such as C/C++. However, the high density of transistors available with today's technology also enables the fabrication of large field-programmable gate array (FPGA) chips. A recent trend is to use FPGA devices to execute simulations of processor models because these devices offer the flexibility of re-programmability with the speed of hardware execution.

FPGA-based simulators have reported and projected speeds two orders of magnitude faster than their software counterparts [72, 8, 54, 19]. There are two approaches for simulating a computer architecture in a FPGA. The first approach is to emulate the entire implementation of the architecture in the FPGA [72]. For the implementation to be useful for performance analysis, it has to, of course, allow for visibility and statistics collection for each run of the simulation. The advantage is that the amount of communication from the FPGA board to the outside host system is reduced. On the other hand, implementing an entire design in FPGA may not be feasible. For instance, a multi-core architecture with 2MB L2 private caches would be hard to fit in FPGAs.

Another approach takes advantage of the functional and timing partitioning present in most simulators, also discussed in section II.C.1. The functional simulator implements the data path, and the semantics of the ISA, to execute instructions. The timing model keeps track of latencies and the control-

path of the structures implemented. For instance, a cache in the timing model does not store any data, but only keeps track of tags, valid and dirty bits. In software simulators, most of the run-time is spent in timing modeling because of the large number of parallel structures to model, and also due to the fact that statistics are tracked. Given this knowledge, it seems appealing to let a hardware (inherently parallel) platform to model the timing, and relinquish the functional modeling to software, sitting on the host platform connected to the FPGA [19, 8, 54]. This greatly simplifies the amount of logic going into the FPGA, enabling it to fit bigger performance models. On the downside, there is a greater cost of communication between the FPGA board and the host platform implementing the functional model. The communication has to be minimized and implemented carefully to avoid large performance overheads. FPGA-based simulation also requires a change to the way architects are used to writing simulators. The timing specification needs to be written in hardware description languages, and not high level languages such as C/C++. Also, there are many tools and simulators already implemented in software used by companies and research groups. The approaches presented in chapters III and IV are applicable to FPGA based simulation as well. Operating system side effects emulation and non-determinism across simulation runs of multi-threaded programs simulated in different configurations are problems to be solved there as well.

II.E Binary instrumentation

Binary instrumentation is a technique to observe the behavior of instructions executed by a program through the insertion of additional code. The instrumentation can be done statically, at compile time, dynamically, during execution, or by modifying the source code.

This section focuses on dynamic binary instrumentation because of its

use to implement the techniques described in chapters III and IV. Furthermore, the description of this section is based on a binary instrumentation infra-structure called Pin [42], from Intel Corporation. Binary instrumentation allows one to observe the architectural state of a process, such as its registers and memory values, as well as control flow information. It allows users to add function calls, called *analysis routines*, through which architectural state can be passed in as arguments.

The instrumentation is performed dynamically, by a just-in-time (JIT) compiler. The instrumentation engine intercepts the execution of the first instruction of the program and translates a sequence of instructions from the original binary (a sequence of basic blocks) into a new sequence of instructions. The program then executes the translated instructions. The translated sequence of instructions is almost identical to the original one, except that the instrumentation engine ensures that, after the sequence is executed, it regains control in order to instrument the following sequence. The translated instructions are put into a code cache. When the code is translated, the instrumentation engine has a chance to insert the instrumentation code, or the analysis routines calls. Eventually, most of the code executed by the program comes from the code cache, along with the analysis routines. Only when new code, un-instrumented by the application, is touched does the instrumentation engine have to execute again. This type of instrumentation requires no recompilation of the application's binaries or shared libraries.

Pin, the instrumentation engine used in this dissertation, offers a rich set of APIs for inspection and instrumentation of the program binary, allowing one to monitor instructions depending on type (memory read or write, addition, multiplication, control flow, etc.), number of operands, and others. Instrumentation can also happen at the basic block, procedure calls or image loading levels. Ad-

ditionally, it allows call-back registration for specific events, such as system calls, signals, and beginning and end of a thread. Pin only instruments the user-code and the shared libraries, not the operating system code.

II.F Summary

This chapter presented an overview of computer architecture simulation and techniques to both reduce and speed-up simulation, which is 4 to 6 orders of magnitude slower than the hardware it models. This chapter is not a complete reference for all existing techniques and simulators, but it aims to motivate the fact that implementing and maintaining simulators requires multi-person-year efforts. Conducting studies based on performance models also present many challenges that are overcome by the techniques described in this chapter. However, there are challenges remaining, which motivated the work presented in this dissertation.

Recently, there has been an effort towards implementing simulation models in hardware, to improve speed and maintain the same accuracy achieved by today's software models. It will take some time, though, until the use of these hardware models become mainstream.

The contributions presented in the following chapters of this dissertation are steps towards helping designers with their tasks of both building and using simulation. The next two chapters delve into the detail of the proposed techniques, along with presenting the results and a more detailed comparison of related work. Chapter V presents the conclusion and directions for further exploration.

III

Efficient Checkpointing for Uni-Processor User-Level Simulation

As previously discussed in chapter II, user-level simulators only perform simulation of the application code and system libraries. These simulators do not simulate what goes on while handling an operating system call or interrupt. Nevertheless, a time consuming part of building such a simulator is correctly emulating the system effects executed as part of the workload under study. For example, the traditional solution [17, 68, 63] to emulate system calls for these simulators is by gathering the required input values from simulated registers and memory state and using them to invoke the call natively. In addition, most of these simulators do not support system effects such as DMA transfers or asynchronous interrupts because of their emulation complexity.

Emulating operating system effects, even just the system calls, can be a tedious exercise. For system calls, the programmer has to be aware of the input and output semantics of every call that needs to be emulated. Apart from

having to handle the complexity of an emulator, porting the simulator to run on a different operating system is labor intensive. Even maintaining a simulator with system emulation can be quite expensive, since the emulator can break when the simulator is run on newer versions of the same operating system. Problems can arise when there are changes to the operating system interface used by the application being simulated because this can require changes to the emulation system. In addition to all these problems, a good number of system effects are non-deterministic in nature, and as a result, emulating them using native system calls during simulation can cause small variations across different simulations of the same program, with the same input. Hence, simulation results may not be completely reproducible.

In this chapter, a technique and a tool that can automatically capture the side effects of all the operating system interactions to support user-level simulation are presented. The tool is called *pinSEL* (Pin-System Effect Logger), which is built using the Pin [42] instrumentation tool. System effects are captured by executing an instrumented version of the binary natively on the operating system for which the workload binary was compiled. The instrumented code creates the *System Effect Log* (SEL) when executed. For each system call executed, the log contains the changes to the register state effected by the system call. The log also contains the values of memory locations accessed by load instructions executed after the system call, if those memory locations were modified by it. The algorithm to identify the registers and memory locations modified by a system call is independent of the semantics of the system call and hence it is easy to implement. The algorithm also allows the implementation to be fully portable across operating systems. The SEL also contains memory values modified by other system interactions such as asynchronous interrupts or DMA, if those modified memory values are accessed by the program being simulated.

Thus the SEL enables deterministic simulation of a program-input pair across system calls, interrupts and DMA transfers. Deterministic simulation is important to accurately compare different alternatives during design space exploration. In addition, pinSEL can also support simulation of multi-threaded applications on uniprocessor systems, which is discussed in section III.C.6.

Using pinSEL, an user-level simulator can avoid the emulation of system effects and the associated complexity. As a result, one can easily simulate real applications from standard operating systems. For example, SimpleScalar [17], which has been widely used for over a decade, emulates just enough number of systems calls to support the simulation of SPEC and similar applications, and cannot support simulation of many real world programs. Using the pinSEL approach, one can now simulate real world Linux applications on an x86 version of SimpleScalar [17], which was modified to consume the logs, without having to emulate any system calls or complex interactions with asynchronous interrupts or DMA transfers. At Intel Corporation, engineers were successful in using the techniques proposed in this chapter to generate the SEL logs to quickly and easily support user-level architecture simulation of MAC OS and Windows applications. Without pinSEL it would not have been practical to port their tool, called *pinLIT*, to support architecture simulation of applications for these operating systems. pinLIT (Pin-Long Instruction Trace) is a tool used at Intel Corporation to gather checkpoints to support architecture simulation.

Using the approach proposed in this chapter, one simulates only the execution of application code and the user level libraries. This is useful for studying applications like desktop and scientific programs, which spend a significant amount of execution time in the user level code. Even interactive applications like `acroread` and `powerpoint`, which spend 80% and 76% of execution time respectively in application code and user level libraries [13], can be captured by

the approach. However, the approach has limitations in that it cannot be used to study applications that are heavily dependent on system interaction (e.g., I/O bound applications like TPC-C [22] and web servers such as DSS that spend significant amount of execution time in the kernel code).

III.A Application-Level Simulation

This section describes two system call logging infrastructures – pinLIT, which is used at Intel Corporation, and SimpleScalar, which is widely used in academia.

III.A.1 pinLIT

An approach used at Intel Corporation, for simulation, is to first use SimPoint [62] to determine representative samples in a program's execution. Then a tool called pinLIT is used to create a checkpoint for each sample. A sample's checkpoint contains everything needed by their simulator to simulate the sample. In this section, this baseline technique used to create a sample's checkpoint is summarized.

SimPoint

The first step is to choose, for a program-input pair, the execution intervals for detailed simulation. SimPoint is used to choose the samples to be simulated. Note that other methods can be used to choose the simulation samples; the selection algorithm is not the focus of this study. A more detailed description of SimPoint is found in chapter II.

Creating Checkpoint Image

Once the simulation points are chosen, the next step is to create checkpoints for each simulation point using pinLIT (Pin-Long Instruction Trace) tool, which is built using the Pin [42] dynamic binary instrumentation tool. The checkpoint and system call tracing mechanism used in pinLIT generates the logs used to guide simulation, as described in the Intel's UserLIT [63] simulation infrastructure.

A checkpoint image for a simulation interval contains all the necessary code and data information that is required for simulating the interval that it represents. This includes a trace of all the input and output values for the system calls executed within the simulation interval.

A checkpoint image for a simulation point is created as follows. The instrumented binary is executed natively and once the execution reaches the simulation point, the processor's architectural register state is copied to the checkpoint. In addition, pinLIT copies all the pages that contain application code and shared libraries to the checkpoint.

For the code and data pages, pinLIT tries to avoid checkpointing the entire data image of the process that exists at the beginning of the simulation point. Instead, pinLIT copies the pages lazily to the checkpoint when they are first used during the simulation interval. This approach avoids logging those data and code pages that are never accessed inside the simulation point and thus reduces the size of the checkpoint. The address locations inside the checkpoint image where the code and data pages are copied to are stored in a table at a particular location in the checkpoint image. This table, called the CheckpointPageTable, is required during simulation to restore the code and data pages.

In addition to copying pages accessed by the program to the checkpoint, pinLIT also logs enough information about the execution of system calls so that

they can be handled during simulation. pinLIT has code specific to each system call that determines the inputs and outputs for every one of them. Before executing a system call, the analysis code in pinLIT logs information about the input values to the system call, along with their address location (for memory operands) or the register name. After the return from the system call, the return value and any memory locations and values modified by the system call are logged. When the system calls are encountered during simulation, the control is transferred to a special system call handler that verifies the arguments and writes the output in the proper memory and register locations. If the input arguments are different, then simulation is halted, since the simulation environment requires and only supports deterministic simulation.

Simulation Using pinLIT's Checkpoint Image

A description of how the simulator uses the checkpoints follows. The simulator first loads the checkpoint image into its address space and starts the program's execution from address 0. pinLIT, when creating the image, inserts special code at that memory location. The inserted code performs basic initialization tasks such as setting up the correct mode of execution for the processor, configuring the status bits for floating point registers, and inializing the virtual address translation engines. This is equivalent to an operating system boot code, and thus it is called a mini-OS. The mini-OS initializes the simulated page table using CheckpointPageTable to map the virtual addresses of the application to the physical addresses where the code and data pages from the checkpoint image are loaded. The mini-OS also registers a system call handler which is invoked whenever a system call is encountered during the program's execution inside the simulator. Finally, the architectural register's contents are read from the checkpoint image and written to the registers. Note that this sets the PC to the first

instruction executed at the beginning of the simulation interval.

When a system call is encountered the system call handler verifies if the system call input values match the checkpoint image values and writes the outputs to the simulated registers and memory. The system call itself is ignored.

III.A.2 SimpleScalar

SimpleScalar supports a system call checkpoint mechanism called EIO (External I/O) logging, which is a trace of the output values of system calls. Playing back the system calls effects from the log ensures deterministic behavior, even if the system call has non-reproducible behavior (e.g. `gettimeofday`).

An EIO file contains a checkpoint of the initial program state that includes memory values and architectural registers that represents the state of the system at the beginning of the simulation interval. The rest of the EIO file contains information about every system call, including all input and output values and the name of the registers and memory address locations where those values should reside.

When the simulator encounters a system call, it restores the necessary register and memory values by reading them from the EIO trace. This method enables deterministic program execution across all the simulation runs.

The mechanism to copy registers and memory values modified by system calls, used by both pinLIT and SimpleScalar EIO traces, is the same mechanism used to emulate system calls during simulation, which is presented in the next section. As it is shown, the mechanism is error prone, complex and very tedious to implement. It is also hard to port when the simulator is run on different host operating systems. As a result, these checkpointing tools do not support the execution of all system calls, which restricts them to creating checkpoints for some programs (e.g. programs that use system calls not handled by the approaches).

In addition, since it is hard to port, it limits studies for applications running only on certain operating systems.

III.B Existing Logging Approach

User-level simulators need to emulate system calls for correct execution of applications. In this section, solutions for emulating system calls are discussed in detail. Additionally, concrete examples llustrate the complexity involved in emulating them.

III.B.1 Emulating System Calls

This section describes in detail how system calls are emulated in SimpleScalar [17]. SimpleScalar's instruction decoder can interpret Alpha [64], ARM [60] and PISA [2] instruction set architectures. Recently, support for x86 ISA was added. For clarity, in the examples of this section, we assumed that an Alpha OSF [46] target binary is to be emulated on a Linux [1] x86 architecture. Note from chapter II that a host is the architecture running the simulation and a target the simulated architecture

Approach

When a system call is invoked by the simulated application, by executing an instruction which triggers it (e.g. x86 `int 0x80` instruction), a special system call handler in the simulator is called. Figure III.1 illustrates the steps for this approach of emulating the system calls. In the figure, the host OS is Linux running on a x86 architecture. The emulated OS is an Alpha OSF, and the binaries are compiled for the same OS and architecture. The numbers with circles represent the steps the simulator goes through. The system call handler's operation can be summarized in three parts.

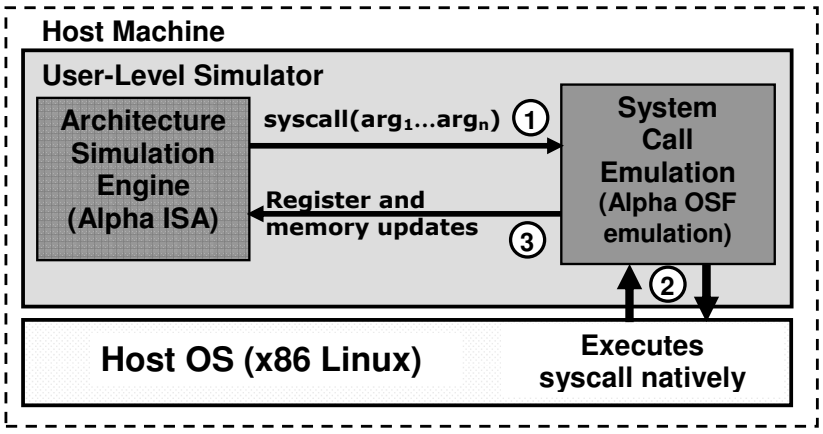

Figure III.1: Traditional emulation of system calls in user-level simulators; 1) System Call is decoded; 2) Memory is allocated in the host OS and the corresponding system call invoked natively; 3) The register and memory side-effects are copied into simulated register and memory respectively.

First, when a system call is found, it invokes the emulation engine. The system call handler, inside the emulation engine, has to decode the system call invoked by the application and obtain the necessary input arguments from the simulated register and memory locations. This is represented by step 1 in figure III.1. Decoding a system call is dependent on the system call identification numbers, which are specific to an operating system. For the Linux operating system, for instance, these numbers are specified in the header file `unistd.h`. This decoding part of emulation should support the operating system for which the application was been compiled for (Alpha OSF in this example). Figure III.2 shows a code snippet extracted from SimpleScalar [17], in which the `switch` statement has a case for each system call emulated.

Second, the system call arguments are used to invoke an equivalent system call, which executes natively on the host machine, represented by step 2

in figure III.1. This part of the emulation should support the operating system in which one wants to execute the simulator, because the arguments to the system call are specific to that operating system. In the example, the simulator (e.g. SimpleScalar) supports execution of Alpha binaries compiled for Alpha OSF systems (supported by the decoding part of the emulator) on x86 Linux systems (on which the emulator natively executes the system calls).

Third, the result values obtained from the native execution of the system call are used to modify appropriate registers and memory locations in the simulator. This is shown as step 3. The emulation engine copies the register and memory side effects back into the simulator, to update its state.

Examples

Consider how the `open` system call is emulated. The system call opens a file returning a file descriptor. The registers mentioned in this section are Alpha [64] registers, the target platform. For the `open` system call, register A1 contains the flag input and A0 contains the address to the location containing the filename. The flag input format can change between operating systems or new versions of the same operating system, which requires changes to the emulation system.

For `open`, the filename is copied into a temporary buffer. The temporary buffer and an integer containing the flag value are used as arguments to invoke the `open` system call natively. The file descriptor returned from the native system call is then copied into the V0 register, or if an error happens the register A3 is set to the error code. Note, the emulation of the `open` system call would be affected if either the binary is compiled for a different operating system or if the host on which the simulator is executed changes.

Consider another example, which is illustrated in Figure III.2. The `read`

```
switch (syscode) {
   ...
   case OSF_SYS_open:
   {
      /* omitted for clarity */
   }
   break;
   case OSF_SYS_read:
   {
     char *buf;
     /* allocate same-sized input buffer in host memory */
     if ( !(buf = (char *)calloc(regs->REG_A2, 1)) )
         fatal("out of memory in SYS_read");

     /* read data from file */
     regs->REG_V0 = read(regs->REG_A0 /* fd */, buf, regs->REG_A2 /* size */);

     /* check for error condition */
     if (regs->REG_V0 != -1) {
         regs->REG_A3 = 0;
     else /* got an error, return details */
         regs->REG_A3 = -1; regs->REG_V0 = errno;

     /* copy results back into target memory */
     mem_bcopy(mem, Write, regs->REG_A1 /* buf */, buf, regs->REG_A2);
     free(buf);
   }
   break;
   ...
}
```

Figure III.2: Code snippet taken from the SimpleScalar [17] source file (`syscall.c`) used to emulate system calls (the code was modified to improve clarity). This source file has over 3500 lines of code to emulate only about 81 system calls.

system call is used to read a specified number of bytes from a file and copy the values read to a buffer. To emulate this system call, SimpleScalar invokes the `read` system call natively using the contents of register A0 and A2 as arguments, where A0 contains the file handle and A2 contains the size of the buffer in bytes. The `read` system call also requires a pointer `buf` to the location where the read contents need to be stored. To accomplish this, SimpleScalar allocates a buffer of size specified by the A2 register and passes the pointer to the `read` system call. Once the system call returns, the contents of the buffer are copied to the simulated location whose address can be found in the A1 register. Finally, the A3 register is written with the error code returned from the native execution of the `read` system call. It also modifies the register V0, with the return value from the system call. Note that, the `read` call can modify the memory location pointed to by A1 and the number of locations modified is dependent on the size specified in the register A2. Thus, it is necessary to capture the system effects on the memory locations.

For this example, A0, A1 and A2 determine the memory locations modified by the system call. Other system calls have different interfaces (e.g. pointers to structures, etc), and each case must be handled individually. The example shown is a simple case. More complex cases are the `readv` system call, which specify an array of buffers or the `ioctl` system call, which is used for a variety of purposes, each one with a special data structure with pointers and specific behaviors. These memory inputs and outputs are system call specific and this is why creating these emulation systems is tedious, error prone, and hard to maintain.

Handling Asynchronous Interrupts and DMA

Emulating more complex interactions with the system through asynchronous interrupts and DMA are even tougher to handle in an execution driven

simulator. It would require modeling the full system including the external peripheral devices. One example of such system is Simics [43]. Hence, applications affected by interrupts and DMA are not supported in the user-level architectural simulators [17, 68, 63]. The logging approach proposed in this chapter captures the memory effects seen during application level execution, which simplifies the execution of such applications significantly.

III.B.2 Benefit of Automated Logging

The above implementation for logging system effects is not desirable for a number of reasons. Note, that handling of system calls involves identifying the input and output values of each system call. This requires decoding and writing code to handle each system call. This method is not portable to simulate applications compiled for a different operating system or even for a different version of the same operating system. In addition, pinLIT and SimpleScalar do not support applications that use asynchronous interrupts and DMA transfers. These issues are solved with an automated system effect logging to capture all forms of system effects, which is described next.

III.C Automatic Logging

In previous sections, we presented a description of how popular cycle accurate simulators [17, 68, 63] need to emulate system calls to achieve correct program execution. For example, SimpleScalar emulates 81 unique system calls to support simulation of SPEC and similar programs. In comparison, the pinLIT simulation tool, used at Intel Corporation, emulates 258 system calls, to support a much wider range of applications compiled for the most popular Linux kernels. Emulating these system effects is tedious to implement, hard to maintain, and error prone.

In this section, an instrumentation tool that can automatically capture system effects in a log is described. This logs can then be used to guide architecture simulation. The tool can also support simulation of multi-threaded programs on a time-shared uniprocessor system, which is discussed in detail in section III.C.6. It can also be extended to support deterministic simulation of multi-threaded programs on multi-processor systems. This is described in chapter IV.

III.C.1 Overview

Instead of emulating the system calls one by one, the idea proposed in this chapter is to automatically capture all the system effects to a program's execution in a *System Effect Log* (SEL). This log file is part of a checkpoint, which can be used to replay the program's execution and simulate it without having to emulate any system effects. The SEL replaces the system effect logging approach used for pinLIT and the SimpleScalar EIO checkpoint trace, described in section III.A. The SEL checkpoints are also used for emulating system calls in the Asim [28] simulator. The logging approach described here is much easier to implement and maintain, and it provides support for asynchronous interrupts and DMA transfers, which are supported neither in the pinLIT nor the SimpleScalar EIO tracing mechanism.

Our system effect logger, named *pinSEL*, uses a dynamic instrumentation tool called Pin [42]. This section briefly describes the key concept that allows one to automatically capture system effects. A straight-forward way to capture the system effects to a program execution is to log the value of every single load instruction executed by the program, and to log the register states and the PC value after handling a system call or an interrupt. Figure III.3 illustrates this approach. Note that all the load instructions executed by the program are

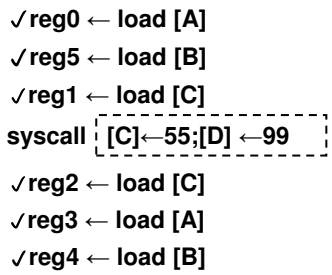

Figure III.3: Instructions executed by the thread. Check marks mean the load value was logged.

logged (indicated by a check-mark next to the instruction). After a system call executes, the processor register values are logged. The system call in the example changes locations C and D. As the program continues executing, the values loaded by the load operations after the system call are logged as well. Hence the values modified by the system call are captured because the value from memory location C is logged and reused during simulation. However, this method is too expensive, both considering runtime and log size overhead. Instead, pinSEL logs a load value only if:

1. The load is the first memory operation to access the memory location or;
2. The memory location accessed by the load has been modified due to a system effect.

The second condition is determined by keeping track of a *user-level* shadow copy of the memory space that is read and written by the application during execution. The redundant copy is called the user-level copy, because it is maintained in the pinSEL's address space, and is updated by pinSEL for load and store operations executed by the application. The user-level copy is *not* updated when the system modifies the corresponding application's memory state while it is handling system calls, interrupts or DMA transfers. Hence, if an

application's memory location is modified due to a system effect, and later a load accesses the same location, pinSEL detects a mismatch between its user-level copy and the corresponding value in the application's address space. When pinSEL detects such a mismatch for a load, it can determine that the program's memory value has been changed by some system event external to the program being profiled, and hence it knows that the load value needs to be logged. A similar mechanism is used to capture the system effects to the register states before and after a system call or interrupt. Figure III.4 shows a representation of the instrumentation tool, which resides in the same address space of the application. Note that when system calls, interrupts and DMA transfers take place, only the actual application memory is updated. Thus, for a program's execution, automatic logging of external system effects to its execution state are logged, without having to explicitly model and emulate the system interactions.

The algorithm used by pinSEL is inspired by the checkpoint scheme used in BugNet [50], for debugging. In their work, a hardware mechanism to log system effects is proposed. Their mechanism uses a bit per word in a cache line, to indicate whether a given word has to be logged during execution. If the bit is not set, when accessing the word in the cache, the word value is logged. This bit is set when the first access to the word happens. After a system call or an interrupt, all the bits are cleared so that load values modified by the system call also gets captured, if they are accessed. This also means that the values that were already logged previously need to be re-logged. BugNet also relies on hardware mechanisms for cache coherence to reset the bits for the words changed by a DMA transfers in order to log its memory side effects. pinSEL does not rely on any hardware mechanism or operating system support. It simply relies on comparing the values in the user's memory space with the values in the shadow copy kept internally. In addition, since pinSEL is a software tool, it can keep

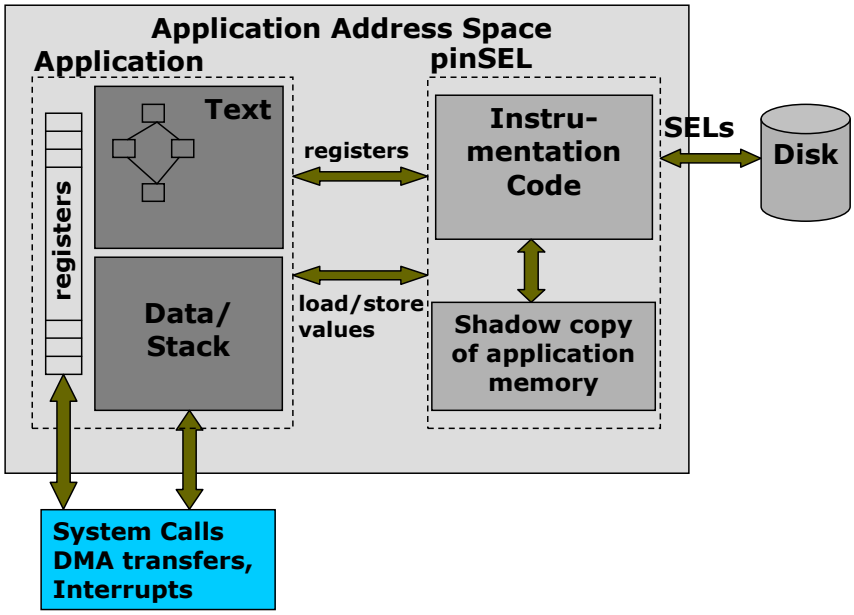

Figure III.4: pinSEL instrumentation tool representation. Both the application and pinSEL's instrumentation reside on the same address space. The instrumentation monitors memory and register values changed by the application. The instrumentation also keeps a shadow copy of the application touched memory.

track of a large number of memory accesses (unbounded if enough memory is available), hence reducing the size of the logs. Finally, BugNet does not provide support for logging of the code executed by the application, whereas pinSEL logs the code as well.

The SELs can then be used to replay a program's execution, to guide architecture simulation and avoid the need to emulate the system interactions. The simulation of application level execution is accurate as SELs enables deterministic replay of program execution. However, there can be small inaccuracies (less than 1% error) while simulating mis-speculated paths if those execution paths access memory locations that have not been logged. This limitation is discussed in more detail in section III.C.8.

III.C.2 Introducing pinSEL

The goal is to collect a *System Effect Log* (SEL) to guide reproducible architecture simulation. The SEL contains the initial register, program counter and memory (code and data) values accessed by the program execution and all the system effects to those memory and register states. In addition, for multi-threaded programs executing on time shared uniprocessor systems, it also contains information about thread interleaving which is discussed in section III.C.6. The SEL can be for the complete execution of a program or just for a sample of program execution. The sample could be hand picked, or chosen using tools such as SimPoint [62].

The SELs are collected by dynamically profiling the program execution using binary instrumentation. pinSEL is similar to pinLIT in that it is used to collect checkpoint traces for simulation. The difference is that in pinSEL the system effects to both memory and register states are captured using a generic algorithm that is completely independent of the operating system. As a result,

pinSEL can easily capture system effects due to interrupts and DMA transfers, unlike pinLIT, which does not capture those effects at all.

III.C.3 Dynamic Instrumentation

To profile a program using pinSEL the program is executed natively on the system that it was compiled for. pinSEL then dynamically instruments the program binary, and the SELs are gathered as the program executes. Pin [42] provides interfaces that allows instrumentation of classes of instructions, specific functions, system calls and interrupt events, allowing the registration of call-backs to our analysis routines at those instrumentation points. When a pinSEL's analysis routine is invoked for an instrumentation event, pinSEL can examine the program's architectural register and memory states, update its internal data structures, and log information to the SEL files if necessary. Then after it is done with the analysis for an instrumentation event, the program's execution continues until the next instrumentation point before invoking an analysis routine again.

pinSEL uses Pin's interface to instrument every load and store instruction, so that the analysis routines can keep track of user-level memory state of application's data sections and capture its initial state and subsequent system effects to them. pinSEL also instruments every basic block. The basic blocks are instrumented to log the initial state and system effects to code regions in the application's memory. One example of code modified by the system is when applications dynamically load a library. When a library is loaded (through a `mmap` call in Linux), it can potentially overwrite code that was already in memory. If pinSEL did not detect that the code changed, when it tried to execute code from the region overwritten, it would execute the incorrect code.

Finally, pinSEL instruments every system call and the user-level interrupt handlers. Pin allows it to register call-backs to analysis routines that are

invoked before and after the execution of system call and interrupt handlers, allowing pinSEL to capture system effects to the register state.

III.C.4 Timestamps

Every log entry in SEL contains a timestamp that tells when that entry has to be used during simulation.

Two types of timestamps are used in the logs - memory operation count and instruction count. The *current memory operation count* of a program execution is the number of dynamic load and store instructions executed since the start of the logging, whereas *instruction count* is the total number of instructions executed since the start of the logging. Tracking instruction count at the granularity of every instruction incurs high instrumentation overhead. Instead, the instruction count is only updated after executing a basic block, where a basic block is a sequence of instructions with a single entry and a single exit point.

The above two counts are tracked only for the application's execution (user code and user level libraries) and are not updated during the execution of the system kernel code. Hence, while simulating the application's execution, these timestamps can be accurately tracked to determine when to use a log entry. To reduce the size of the timestamp being logged, instead of logging the full memory and instruction counts as the timestamp, the logs sizes are optimized by just logging the difference between the prior count and the new count for the current log entry.

III.C.5 System Effects Log Files

A SEL for a program's execution is composed of the following three log files, at a minimum.

- **Code Update Log** - The purpose of this log is to record the initial memory values of the code regions and the system effects to them. This ensures that programs using self-modifying code and dynamically loaded libraries can be handled. Each entry contains (a) an instruction count and (b) the code contents of a basic block and its size. During simulation, when the number of instructions simulated is equal to the instruction count of the next log entry to be used, the logged code for the basic block is restored to the simulated memory before executing the next instruction. The effective address for restoring the code log entry is the simulated program counter (PC) value.

- **Data Update Log** - The purpose of this log is to record the initial memory values of the data regions and the system effects to them. Each entry contains (a) a memory operation count, and (b) the value of a load operation. During simulation, before executing a load operation, if the simulated memory operation count is equal to the memory count of the next entry in the log, the logged value is restored to the simulated memory. The effective address used to restore the log value is the effective address of the simulated load, which can be determined during simulation and hence need not be logged.

- **Register Update Log** - The purpose of this log is to record the initial states of the architectural register values and the program counter values, and capture subsequent updates to them due to the execution of both synchronous interrupts (system calls) and asynchronous interrupts. At the beginning of execution, the initial values of all the registers and the program counter are copied to this log. Then an entry in the log is created whenever an interrupt is encountered with the following information: (a) the instruction count, (b) the value in the program counter before the execution of the interrupt, (c) the sequence of register values modified by the interrupt handler along

with the name of the modified registers, and (d) the program counter value after the execution of the interrupt, if it had been modified. The instruction count along with the PC value, before the execution of the interrupt, together accurately capture the time at which the interrupt was executed during the program's execution. During simulation, when it is time to use an entry from this log, the logged register values are restored in the corresponding simulated target registers. Also, the logged PC value is restored to the simulated program counter if it was also logged. Note, any memory value updated by the interrupt is logged in the Code and Data Update Log.

In addition to the above logs, SEL also records necessary information to simulate multi-threaded programs on a uniprocessor system. This is described in section III.C.6. The following describes how each of the above logs is created in more detail.

Code and Data Update Log

To capture changes in memory due to system interaction, pinSEL maintains a data structure called the *UserMemState*. The UserMemState keeps track of the values for every memory location accessed by the application. The values in UserMemState are updated only for the load and store instructions executed by the application and not by the system code. This is because only the application is instrumented. Thus, it keeps track of what it is called *user-level* memory state. It is essentially a hash table, indexed by the effective addresses. Each entry in the table mirrors 4KB of application's address space. The initial value for each address location in the table is set to zero.

Each load and store is instrumented to keep track of data values in UserMemState. To keep track of code regions in the application's address space, the program's basic blocks are instrumented.

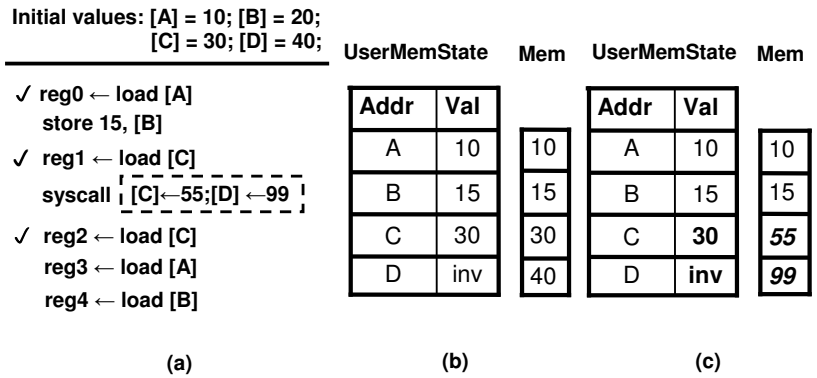

Figure III.5: Example of pinSEL's mechanism to log system effects. (a) - Instructions executed by the thread. Check marks mean the load value was logged. (b) - UserMemState and actual memory state right before executing the system call; (c) - UserMemState and actual memory state after executing the system call and before executing any other instruction.

Analysis for Store - Whenever the application executes a store to an address in its address space, the value in UserMemState is updated for that address with the store's output value. Stores are instrumented before their execution to obtain the effective address, and after to obtain the value from memory.

Analysis for Load - When executing a load, the value in the application's memory for the load's effective address is checked against the corresponding value in the UserMemState. If they differ, then it implies that, (a) it is first time that memory location is being accessed or (b) the accessed memory location has been modified by the system while handling a system event or (c) the memory locations was evicted out of the UserMemState structure[1]. In addition, if the application's store instruction modifies a memory location, it would have correctly updated the UserMemState when the store was executed. Therefore, whenever the load value is different from the corresponding value in the UserMemState, its value is logged in the Data Update Log along with the current memory count. This ensures that the initial memory data values as well as system effects to them

[1]For the implementation presented in this chapter, the size of the UserMemState structure is unbounded. However, if the memory footprint of the application is too large, the structure may need to have a fixed size. In that case, entries in the structure can be replaced by other entries, similarly to a hardware cache structure.

are captured in the Data Update Log. Loads are only instrumented before their execution, to read the value from memory that will be loaded.

When the value for the load is logged because it differs from the value in UserMemState, the UserMemState's value for the load's address is updated with the new value observed in the application's memory. This is required to make UserMemState consistent with the new value found in the application's memory state, so that future loads to the same location will not result in additional logs, unless it gets modified due to a system effect.

Figure III.5-(a) shows a sequence of instructions executed by a thread. A `syscall` instruction invokes a system call, which changes the memory values as shown in the dotted box. Figures III.5-(b) and III.5-(c) show the state of the UserMemState, along with the actual memory state, represented by `Mem`, at different points of the execution. The initial values for the hypothetical memory locations accessed are also shown in the figure. Figure III.5-(b) shows the state of UserMemState right before executing the system call. Values for memory locations A and C are logged, because they are the first access to those locations. Value for memory location B is not logged because the access was a store. Note that the values in the UserMemState for A, B and C match the value in memory, because the instrumentation code updated the UserMemState structure. Location D has not been accessed yet, hence it shows an inv value, representing an invalid value, which could be just zero. After the system call executes, memory locations C and D are modified with values 55 and 99 respectively. Figure III.5-(c) shows the state of UserMemState and the actual system memory. The values in UserMemState are not updated because the system call code is not instrumented by pinSEL. This means that when the first instruction after the system call executes, loading a value from C, pinSEL will log it because it mismatches the value in the actual memory. The following two loads will not be logged be-

cause they are not the first accesses to those locations and the values match the ones in memory. Even though location D was modified by a system call, it was never logged because the user code never touched location D. Hence only system call side effects used by the application are logged.

Analysis for Basic Block - The Data Update Log created by analyzing load and store instructions captures the initial memory values and system effects to only the data values accessed by the application. It does not contain the instructions fetched for execution, unless they are loaded by some load instruction.

For the binary instrumentation analysis, an instruction fetch can essentially be treated as a load from the address specified by the program counter value. To capture the code, every basic block in the program is instrumented to register a call-back routine that is invoked before the execution of each basic block. A basic block is a sequence of instructions with a single entry and a single exit point. Hence if the program control reaches the beginning of a basic block, all the instructions in the basic block are executed.

When the analysis routine is invoked just before the execution of a basic block, the N bytes of application's memory values at the location specified by the program counter value are checked against the corresponding values in the UserMemState. If the comparison fails for any of the bytes, the value is logged in the Code Update Log along with the instruction count. Also, the value in UserMemState is updated with the up-to-date value in application's memory in order to make them consistent.

During simulation, when the simulated instruction count equals the instruction count in the next code log entry to be consumed, the code from the log is restored to the simulated memory using the address in the simulated program counter.

Handling Self-Modifying Code and DLLs - The code logging mechanism handles applications using self-modifying code. The Code Update Log captures the initial values in the code regions during execution. An application using self-modifying code modifies itself by executing store instructions, which will be deterministically replayed during simulation. That is, one knows the exact input and output values for each store instruction and hence handling self-modifying code is not an issue.

It can also handle applications using dynamically loaded libraries (DLLs). A dynamic library can be loaded during a program sample's execution through the invocation of a system call (eg: `mmap` system call in Linux). Since the Code Update Log captures any changes to the code regions, it will also capture the contents of the dynamically linked libraries when they are fetched from memory for execution.

Note, a more light weight approach (in terms of run-time overhead) for logging code is to integrate the code logging with the run-time system used to execute the program, instead of instrumenting every basic block. Using this type of run-time system, no code can execute until it has first been pre-processed. The first time it is executed, it will be analyzed once and the code to be executed will be logged. Then the code will not have to be analyzed for logging again, unless it is modified. If there is self-modifying code, then the code would be invalidated by the run-time system. It will then be re-analyzed before it can execute again, and when it is, it will be re-logged. The run-time system to support this type of approach could be a virtual machine or a dynamic binary instrumentation system like Pin. Recall from section II.E that instructions are first translated to a code cache before they start execution. The logging would take place when this translation happens. Since the code is analyzed for logging only once before it is first executed, the run-time overhead will be minimal compared to instrumenting

every basic block.

Register Update Log

The Code and Data Update logs described in the previous section can capture initial memory values and the system effects to them. In addition, the initial register and program counter (PC) values need to be logged, and system effects to them due to the execution of a system call or an interrupt handler.

At the beginning of the execution of a sample, the initial values of the architectural registers and the program counter are copied into the Register Update Log. Thereafter, an entry in the log is created whenever the program execution encounters a system call or an interrupt.

Pin [42] provides APIs to register call-backs to analysis routines before and after the execution of system call (synchronous interrupt) and signal (asynchronous interrupt) handlers called *SIGNAL_BEFORE_CALLBACK* and *SIGNAL_AFTER_CALLBACK*.

Before the execution of a system call or an interrupt, the state of all the application's architectural register values and the program counter value (which are accessible through Pin's interface) are logged in pinSEL's internal data structure. Then immediately after the execution of the system call or the interrupt, the current register and PC states are compared with the recorded values. The values of the registers and PC for which the comparison fail are logged in the log entry. This eliminates the need to know the system call register use conventions and intricacies for the operating system and the architecture where the tool is running.

Each log entry also contains the instruction count and the program counter value before the execution of the system call or the interrupt. These two values, together constitute a timestamp that tells when the log entry should be

used during simulation. During simulation, the next log entry from this log is used, if the simulated instruction count is greater than or equal to the logged instruction count, *and* if the simulated program counter value matches with that of the logged PC value.

III.C.6 Simulating Multi-threaded Programs on Uniprocessor Systems

The approach described also allows simulation of multi-threaded programs on uni-processor systems. For each thread, a SEL consisting of Code, Data and Register Update log is created and all the data structures in the pin-SEL used to create these logs are kept private to each thread. Whenever a new thread is created within a sample's execution, a Register Update Log for the thread is created and the thread's initial register and program counter values are logged. Thereafter, for each system call or interrupt executed as part of the thread, a new log entry in the thread's Register Update Log is created. As memory instructions are executed, the Data Update Log for each thread is also populated with memory values

During simulation, the thread inter-leavings are simulated just as they would occur on a uni-processor. To achieve this one needs to capture context switches and log sufficient information about them in a *Context Switch Log*. This log file is shared among all the threads in the program's execution and is created as follows. Whenever there is a context switch from one thread to another, an entry in this log is created. Context switches between the threads of the profiled application are detected as follows. Pin internally keeps track of a unique thread ID for each thread and these IDs are accessible from the analysis routines. Inside each analysis routine, the current thread ID is compared against the thread ID seen by the last executed analysis routine. If they differ, then it means that there

was a context switch.

On detecting a context switch, an entry in the Context Switch Log is created. The entry contains the thread IDs of the thread that is context switched out and the thread that is context switched in. Also, the log entry contains the memory count corresponding to the last memory operation executed by the thread that is context switched out. While simulating a thread, its memory count count is tracked, and if it equals to the memory count for an entry in this log, it means that the thread needs to be context switched out. From the logs, it is also known which thread should start simulation next. The above mechanism is useful for simulating multi-threaded programs on uniprocessor systems by reproducing the thread inter-leavings.

III.C.7 Atomic Analysis

In section III.C.5, the analysis functions that can automatically capture the system effects to memory are described. To record the Data Update Log, pinSEL's analysis routine compares the load value with the value in UserMemState when executing every load. However, between the execution of the analysis routine and the application's load, there can be another thread that modifies the memory value accessed by the load. For this to happen, a thread has to be context switched out after the analysis routine executes, and another thread that modifies the value context switched in. Once the original thread context switches in again, the value seen by the analysis routine can be different from the value that is actually loaded by the instruction. Essentially, the execution of the application's load and the analysis routine is not guaranteed to be atomic. Figure III.6 illustrates the problem. Thread 1 executes a memory operation reading a value from memory location A. This memory location is initialized to 0. Its analysis routine, executed before the instruction executes, knows the effective address and

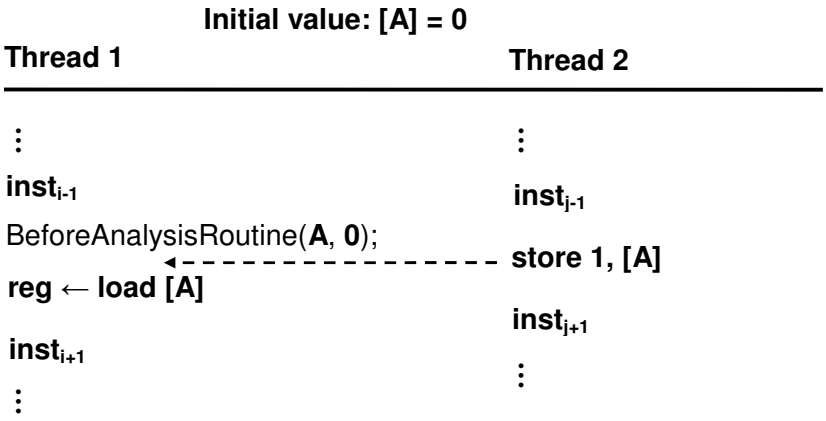

Figure III.6: Atomic analysis problem. Thread 1 executes a load instruction. Its analysis routine sees the instruction loading 0. However, thread 2 changes the value after the analysis routine reads the value. As a result, thread 1's instruction actually reads the value 1. Hence the value logged is incorrect.

the value that will be read by the instruction, which is 0 in the example. However, before the load instruction executes, thread 1 is context switched out, and thread 2 context switched in. Thread 2 then modifies the value to 1, by storing to the same address. Consequently, the analysis routine sees the load reading the value 0 but the actual instruction reads the value 1, which is incorrect. Note that if the instrumentation happened after the instruction executed, the problem would *not* be solved. The same exact problem could happen between the execution of the instruction and the analysis routine executed after.

This is not a problem for single threaded SPEC programs, but it needs to be handled for multi-threaded programs with shared memory interactions. Programs with asynchronous interrupts do not suffer from the same issue because the instrumentation engine (Pin [42]) only delivers interrupts at the end of a basic block. Therefore the interrupt never happens between the analysis routine and the instruction. This atomicity problem is solved by doing the following. Every every memory operation is instrumented before and after its execution. A lock is acquired when executing the analysis routine before the instruction executes and released when executing the analysis routine after the instruction executes. To

minimize contention for the locks, a lock per address range is implemented. Note that the locks to guaranteed atomicity are only needed for multi-threaded programs. When capturing single-threaded programs, the locks are not necessary, and hence are not implemented. This reduces the overhead in instrumentation time because it avoid acquiring/releasing the locks, as well as eliminates the need to execute analysis routine code after most memory operations (to release the locks). A more efficient solution can be implemented by rewriting the memory access instructions. This could be done by the binary instrumentation engine when instrumenting instructions. For example, for a load instruction, the binary instrumentation engine re-writes the instruction such that the values are first copied to memory local to the thread. This memory is then used by the instruction and by the analysis routines, hence guaranteeing that the same values are seen by both. For write instructions, the memory is written to the shared memory area as well as a local memory area. The analysis routines then accesses the data from the local memory. However, instructions that perform read-modify-write operations still need locking to guarantee atomic behavior. Nevertheless, the amount of locking could be reduced significantly.

III.C.8 Architecture Simulation

pinSEL's logging approach replaces the pinLIT logging approach for system effects and SimpleScalar EIO traces to deterministically guide the program's execution through simulation. The above sections describe how and when to use each of the logs to guide simulation. A version of SimpleScalar which runs x86 binaries has been implemented and modified to consume pinSEL logs to guide simulation. At the University of California, San Diego, pinSEL is currently used to collect SELs for Linux applications which can then be used for simulation in SimpleScalar-x86. At Intel, pinSEL's approach of logging system effects has

been used by pinLIT for Linux, Mac OS, and Windows applications, to guide architecture simulation. Recently, the pinSEL tool has also been used to guide architecture simulation on the Asim [28] simulator.

Advantages of pinSEL

The main advantage of using the SELs described thus far is the ability to automatically log system effects to avoid emulation of system calls in simulators. Another advantage of using our pinSEL approach is that the simulator can easily support the simulation of applications compiled for any operating system as long as Pin [42] can support it.

The other advantage of using SELs is that it provides deterministic re-execution of the program to guide simulation. Since the same SEL is used across all simulation runs, the load instructions read exactly the same values and hence the execution of the program follows the same path in all the simulation runs. This is an important property for simulating user interactive applications and applications whose behavior depends on the events coming from the external world. One example of such application is a web server, whose behavior depends on the content as well as the network latencies observed.

Limitation

Simulations based on checkpoints and traces can be affected when it comes to simulating the wrong path (mis-speculated path) in the program's execution. When simulating using pinSEL logs in SimpleScalar, it is still possible to model the wrong path execution similar to execution driven simulation. However, there could be small inaccuracies in the simulation, if the wrong path of execution tries to execute code or data that was neither logged nor regenerated during simulation. In such an event, the wrong path execution can either stop

fetching down the mis-speculated path or it can proceed by consuming a null value.

Van Biesbrouck et.al. [14] examined this issue for their technique to reduce the checkpoint sizes which is described in section III.F.3. Their simulation uses optimized checkpoints that contain only the code and data addresses used during the sampled simulation interval and as a result experience the same problem as pinSEL guided simulation does. However, they found that the error in performance metrics due to the above inaccuracy is less than a 1% on average. Moreover, this inaccuracy is consistently biased in one direction across different simulation runs for an application while exploring the architecture design space, enabling a fair comparison across different design alternatives.

Finally, pinSEL's utility is limited when one wants to use it to study applications whose performance is heavily dependent on system interaction. For example, one would only be able to capture less than half of the execution of I/O bound applications like TPC-C [22] and other server applications like DSS (Darwin Streaming Server), since they spend so much time executing in system code. However, there are many interactive desktop applications like `acroread` and `powerpoint`, etc, which spend 76% to 80% of their execution time in application code and user level libraries [13] (non-kernel code), which can be captured with our approach. This makes pinSEL useful to evaluate these types of applications.

III.D Logging Results

In this section the runtime overheads in collecting pinSEL logs along with the log size overheads for all of the SPEC programs and a handful of desktop interactive programs are examined. The execution of all these programs was traced using pinSEL. The generated logs were then used in x86 SimpleScalar to

guide simulation. The logs were also used to guide simulation in the Asim [28] simulator, developed by Intel.

III.D.1 Benchmarks

For evaluating and verifying the logging approach all the SPEC2000 programs with reference inputs were run under pinSEL to collect logs for simulation. A handful of desktop interactive Linux programs were also run. For the interactive applications each program was run natively for about two minutes. After that the same actions were replayed with the program instrumented with pinSEL to gather the logs.

The programs we examined were `xpdf`, `acroread`, `ggv`, `xv` and `rdesktop`. The first two are used to read PDF documents and `ggv` is used to read Postscript documents. `xv` is an image processing application. `rdesktop` is used to remotely access a Windows system. For the first three programs, files, were open and read, browsed through, enlarged, and exited. `xv` was ran and used to open two JPEG images, zoom in and out on an image and run an image sharpening utility that comes along with the application over those images. For the `rdesktop` program, a connection was open to a windows machine, and a few web-sites were browsed using `firefox`. Also `power-point` was opened and worked on before logging out.

III.D.2 Avoiding Software Complexity of System Effects Emulation

One important result of the approach presented in this chapter is that to collect logs and to simulate a wide variety of applications, including real interactive programs, no system emulation support needs to be provided. In comparison, Intel's pinLIT has a large body of switch-case statements to handle each of the 258 different system calls and it still can simulate only a limited variety of applications. x86 SimpleScalar has support for emulating only 81 different system

calls which is the set of system calls that are sufficient to simulate the SPEC workload. However, this support is inadequate to simulate interesting desktop applications.

Using the automated pinSEL logging approach, one can now simulate any type of application in the x86 SimpleScalar and Asim simulators. Also, since the application level simulation approach is independent of the operating system, it ensures the portability of the simulator to any version of Linux operating system. This means that the logs can be collected in any machine were Pin works, and simulated in those simulators. The approach is easily portable to other operating systems, as long as there is a binary instrumentation tool that can allow us to collect SEL files, which is why at Intel, pinSEL has enabled simulation of Windows and MAC OS based applications.

Though the mechanism can enable simulation of any application in x86 SimpleScalar or Asim, as pointed out in section III.C.8, the evaluation will be meaningful only for those applications that spend a significant proportion of execution time in the application and user level shared libraries.

III.D.3 Log Sizes and Logging Overhead

This section examines the runtime and space overhead in generating SEL logs for the full execution of the SPEC2000 programs. For this, the SPEC2000 programs were run over the reference inputs with the pinSEL tool to generate the logs. The results are averages for each program ran with all its reference inputs. Figure III.7 shows the number of dynamic instructions and the number of dynamic memory read instructions executed by the SPEC2000 programs. On average, there are 119 billion dynamic instructions and 30 billion dynamic memory read instructions.

Each entry in the hash table structure (UserMemState) keeps track of

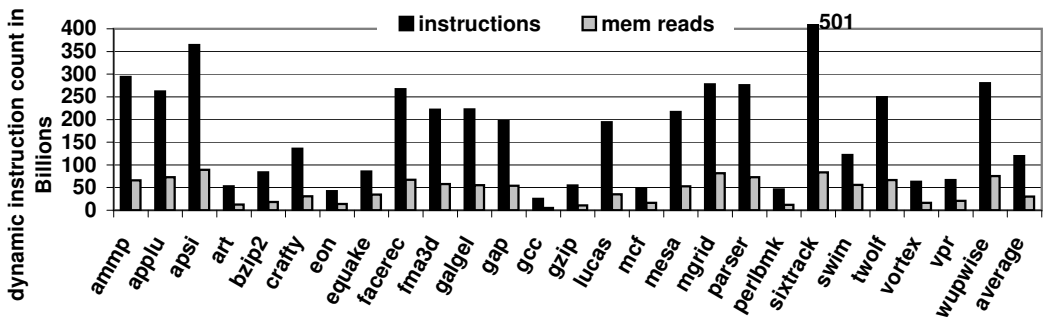

Figure III.7: Number of dynamic instructions and dynamic read memory instructions for the SPEC2000 programs examined.

4KB of data or code. Figure III.8 shows the slowdown for running the program with the pinSEL tool. The runtime overhead is with respect to natively executing the workload. The figure breaks down the slowdown in three categories. The first (*pin*) measures the underlying slowdown of the binary instrumentation system. This is the slowdown for translating instructions into the instrumentation engine's code cache and executing them from there. This overhead is about 1.5x. The second slowdown is the overhead of a memory and basic block profiling (*mem-prof*), which are two types of instrumentation heavily used by pinSEL. This measures how much overhead is present from invoking the analysis routines to profile basic blocks and memory operations. The instrumented instructions and the intrumentation points (e.g. before the instruction, after the instruction, etc) are the same instrumented by pinSEL. The code executed by the analysis routines is much simpler though. For the basic blocks, it increments a counter with the number of instructions executed in that basic block. For memory operations, it counts them and breaks the counts down per category, where the categories are: read or write of one byte, read or write of two bytes, read or write of a word, and read or write of multiple bytes. This resembles the way pinSEL keeps track of memory operations and hence measures the overhead of invoking the analysis routines for each case. This overhead is on the order of 19x on average. This

is the minimum overhead to just execute the basic profiling needed by pinSEL. The remaining overhead is due to pinSEL's analysis routines code, to look up the UserMemState structure and log the data if needed. This overhead is about 71x. This overhead depends on how many instrumentation instructions are executed on behalf of each memory operation. It roughly depends on the number of memory operation instructions in the program, and also on the type of memory operation (the categories mentioned above). The worst-case total runtime overhead is about 135x for `galgel` and `perlbmk`. On average, slowdown experienced is about 92x. These overheads are for tracking the full execution of the program. Also, notice that the runtime overhead due to instrumentation for programs that usually have low IPC (eg: `mcf`) is only in the order of 15x. Whereas, programs with high IPC experience slowdowns in the order of 120x to 130x (`gap`, `perlbmk` and `vortex`).

Figure III.9 shows the log sizes for capturing the SEL for the full execution of SPEC2000 programs studied. The results show log sizes with and without compressing the logs using bzip2, ran with the default compression level. In the worst case, it requires about 235MB of un-compressed SEL to capture the full execution of `fma3d`, and only requires 9MB of SEL after compressing it using bzip2. On average, it requires only about 24MB of uncompressed SEL, which when compressed requires 2.5MB of disk space to capture the full execution of a SPEC2000 program running the reference inputs. The size of the SEL files is dominated by the Data Update Log.

The sizes of SELs are dependent on the number of system calls executed and are also heavily dependent on the amount of data read from the system through those system calls. Since `fma3d`, a mechanical response simulation program, reads a large amount of data (a large number of finite elements for its simulation) from its input files through system calls, it incurs a large log size

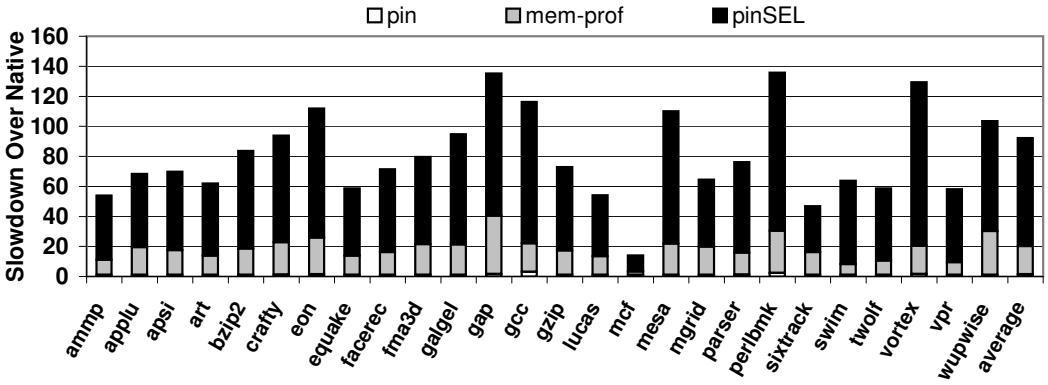

Figure III.8: pinSEL logger runtime slowdown (number of times, not percentage) over native execution for the SPEC2000 programs.

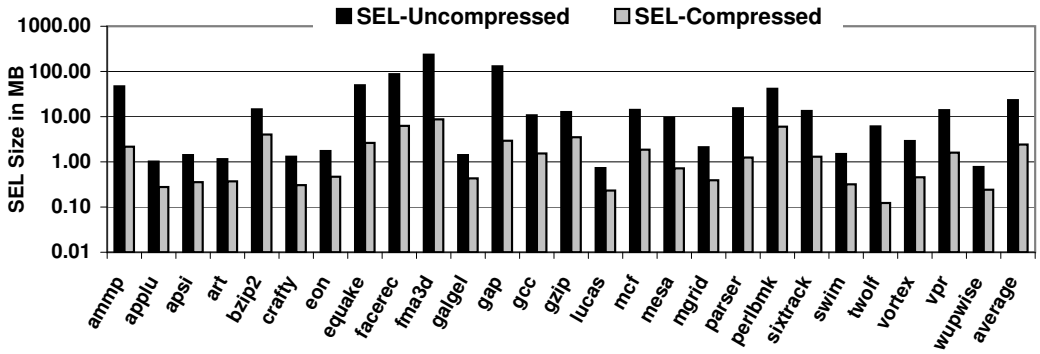

Figure III.9: pinSEL log sizes to capture the full execution of the SPEC2000 programs, with and without compression using bzip2.

overhead.

Note that only the load values touched for the first time or modified by system calls are logged. For logging the data touched for the first time, UserMemState is analogous to a cache structure, and hence only the misses due to cold start are logged. Since pinSEL uses a unbounded number of entries in the UserMemState data structure, the number of load values to be logged is very small. In addition, all the data that is written to first, before being read, does not need to be logged, because it can be generated during simulation. As a result, only 0.01% of the load instructions result in log entries, which explains the small size for the logs. On average, only 56KB for every 100 million memory instructions executed by the program.

Figure III.10 shows the total number of system calls executed in the SPEC2000 programs studied. On average, there are about 6,700 system calls executed during the full execution and in the worst case for `sixtrack` there are about 104,000 system calls being executed.

III.D.4 Log Sizes Per Simulation Point

It is a common practice in computer architecture to choose representative samples of program execution [62] and perform detailed simulation only for those samples to save simulation time. Chapter II discusses SimPoint and other possible techniques. Hence, a quantitative analysis of the SEL size overhead for capturing an arbitrary sample of program execution is presented.

To quantify the average SEL size for an arbitrary sample, each program's execution is broken in 100 million consecutive intervals (samples). For a program, a SEL is collected for each interval. To create a SEL for an interval, all the entries in the pinSEL's data structures (e.g., UserMemState, register values, etc) must be cleared at the beginning of the interval of execution. This will ensure that

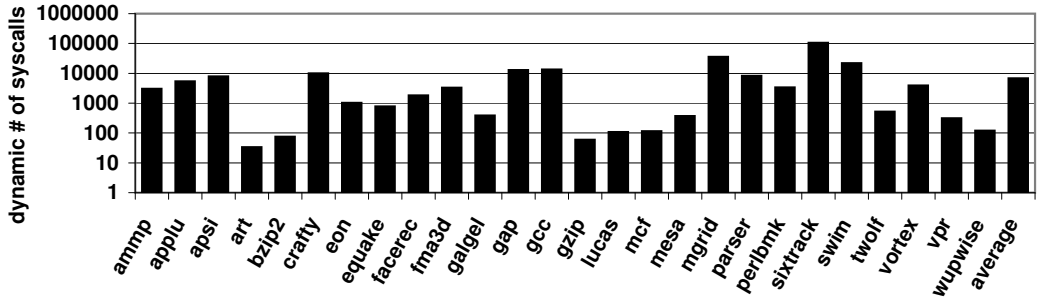

Figure III.10: Number of system calls executed in SPEC

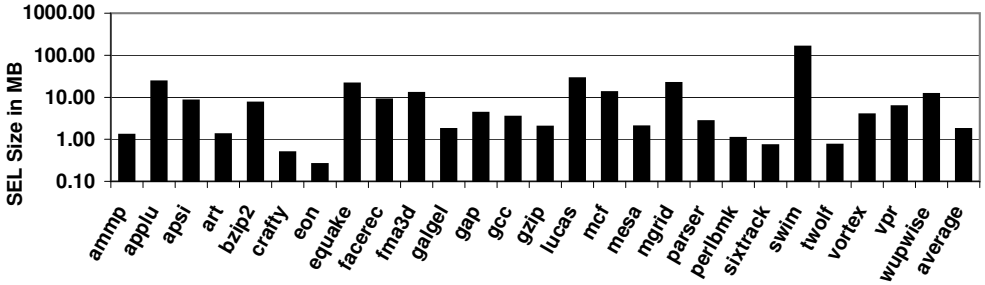

Figure III.11: SEL size required to capture a simulation point of 100M instructions for each SPEC program on average, without compression.

the SEL captured for an interval of execution (sample) has all the information to simulate the program's execution starting at the beginning of that sample. Thus, using a SEL for a given sample enables the simulation of only the 100 million instructions in the sample.

Figure III.11 shows the average SEL size (without compression) for 100 million instructions of execution for each program. The SEL size shown for a program is an average of the SEL sizes of all 100 million intervals for the program. The results show that on average, it requires 1.75MB of SEL to capture a program's 100 million instruction sample. In the worst case, for `swim`, it requires 159MB of SEL to capture a sample of program execution (100 million instructions). But as shown in Figure III.9, the SEL size required to capture the full execution of `swim` is only 1.5MB. Similar results, can be observed for `applu`, `apsi`, `mgrid`, `mcf`, `wupwise`, `ammp`, `art`, `galgel`, `vortex` and `lucas`.

The reason for this difference is that for an arbitrary sample, all the data live (touched by the application for the first time in the sample) coming into the sample's execution which are used (through a load) before getting redefined (through a store) during the sample's execution needs to be logged. In the worst case, this could potentially be every single load executed in a sample of execution. In the case of SPEC2000, an average of 3.5% of the memory read values are logged, when collecting checkpoints for samples of 100 million instructions. In comparison, when one starts generating a SEL from the start of execution, the only data that is live to that SEL is the data read from the global data segment and input data files used during execution. In addition, the data brought into memory by system calls is also logged. All the other data generated by the program itself during its execution is not logged when generating one SEL starting from the beginning of execution. This is the reason for the average sample size for a SEL being larger than the size of a single SEL generated for the full execution. A similar observation was made by Bronevetsky *et al* [16] in their design of a checkpoint and recovery system. They choose to create a checkpoint of the application's state during program execution when the amount of live data is smaller, so that the resulting checkpoint size is also smaller.

III.D.5 Log Sizes for Non SPEC Programs

Logs were also gathered for a few interactive desktop applications which can now be simulated easily in x86 SimpleScalar and Asim. Each of these interactive programs was run for a few minutes performing some common tasks. Figure III.12 shows the average SEL size (with compression) required to capture 100 million load instructions. On average, it requires about 0.4MB of compressed SEL to capture 100 million load instructions for these interactive applications. The average number of load instructions executed between two system calls or

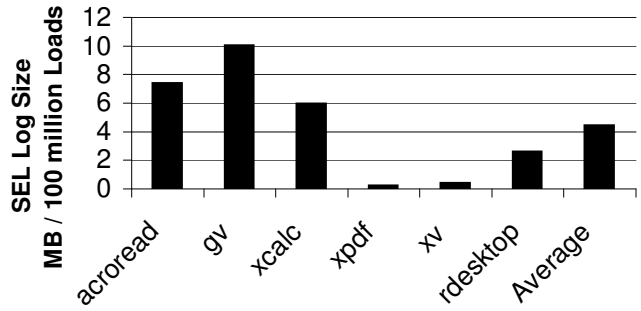

Figure III.12: SEL size required to capture 100 million load instructions for interactive desktop applications with compression.

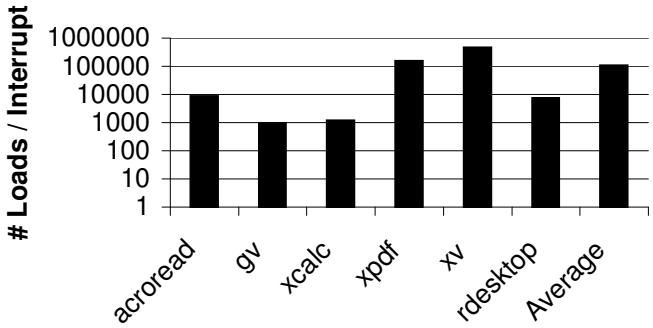

Figure III.13: Average number of loads executed between two interrupts (including system calls and asynchronous interrupts).

interrupts in the Figure III.13, for these applications, is also shown. It varies from 1000 load instructions for `gv`, which incidentally also requires the largest SEL size, to about 460,000 load instructions for `xv`, which requires a smaller SEL size.

III.E Other Uses of pinSEL Checkpoints

The logging scheme presented in this section has proved to be very useful when generating checkpoints for simulation. By using the checkpoints proposed, one can avoid the emulation of the system interactions completely, including system calls, DMA transfers and interrupts, and guarantee deterministic results across simulations of uni-processor architectures. The checkpoints are useful in

other contexts as well.

The checkpoint log files can be viewed as a mechanism to compress the dynamic execution of the program. The minimum amount of information is stored in the logs, and the rest of the information is re-generated during execution of the program. The simulator can be viewed as one possible tool to decompress the logs and to retrieve the instructions to be executed. This does not mean that the simulator is trace-driven. It just means that a subset of the instructions and the data read from the program simulated comes from the logs. The simulation is still execution driven, and complex mechanisms such as wrong-path behavior can still be modeled.

In addition to the simulators modified in this work, another tool to decompress the logs was also developed. In particular a tool also based on binary instrumentation, named *pinPLAY* , because it is also based on Pin [42]. This tool reads the initial code, data, and register states from the pinSEL logs. Once these are loaded, the binary instrumentation engine starts fetching instructions, translating them into the code cache and executing them. Because the instructions are executed natively, after the binary translation takes place, the execution of the program from the pinSEL checkpoints is fast. Executing under the binary instrumentation engine, also enables one to use almost all the APIs provided by it[2], in order to analyze and print out any information about the program being executed. As a result, this tool can be used to generate traces for trace-driven simulation as well. One example is generating memory access traces, or branch outcome traces. Such framework for tracing and analysis of programs is similar to the ones presented by Bhansali *et al* [12] and Xu *et al* [78].

The pinPLAY tool was also extensively used to help debugging the implementation of the simulators which consume the pinSEL logs. pinPLAY was

[2]The APIs that use the binary debug section, with all the symbol information, do not work because only parts of the binary are available from the log

used to generate a trace of committed instructions, which is then compared to the trace of committed instructions generated by the simulator. If these traces match, it means that the modifications in the simulators to consume the pinSEL checkpoints are correct.

And finally, at Intel, the pinPLAY tool is used to integrate the checkpoint generation mechanism with the sample selection mechanism [53]. In particular with SimPoint [62]. SimPoint requires a two-pass approach for selecting checkpoints for simulation, as described in chapter II. One pass is needed to profile the program (e.g. collect basic block vectors) behavior. From these profiles the simulation points are selected. Another pass is needed to "visit" the simulation points and generate the checkpoints, which is done by the pinLIT (defined in Section III.A.1) tool. This two-pass process is executed by using binary instrumentation, due to its easy of use and speed. Visiting the simulation points requires that the executions paths taken by the profiled run and the checkpointing run to be the same. This is required to make sure that the profiled behavior is the same behavior captured by the checkpoint, and also to be able to find the regions of interest [53][3] However, this is specially hard to achieve for interactive applications, but also a problem for non-interactive applications, because the behavior of certain system calls is non-deterministic in nature. Using pinSEL checkpoints for the full runs along with pinPLAY solves this issue, because the runs under pinPLAY are guaranteed to be deterministic. In this way, the run that profiles the program, also generates pinSEL checkpoints for the entire run. The simulation points are selected based on the profiles. The second run, to create the checkpoints for each sample, are executed from the pinSEL checkpoints, which is deterministic and guarantees the same execution paths. Then for each of the simulation points selected, a new pinSEL checkpoint is generated with only

[3]A common practice to find the region for simulation is to use a program counter and the number of times the program counter appears in the execution to mark the beginning of a region. With slight variations from run to run, these markers may not be found. Under pinPLAY this problem is solved.

the information for that sample.

III.F Related Work

This section discusses existing solutions to handle system effects.

III.F.1 Handling system effects for User-Level Simulation

Many popularly used cycle accurate simulators [17, 68, 63] simulate just the user code and this is sufficient for studying many micro-architecture level optimizations and design choices using workloads like SPEC. However, even though their goal is to simulate only the user code, they still have to emulate the system calls to obtain correct execution of the program. The conventional solution to emulate system calls is to decode the system call and obtain the arguments. Then using those arguments the simulator invokes an equivalent system call that can be executed natively on the host machine on which the simulator is executing. The result values obtained from this native execution are then used to modify appropriate simulated registers and memory locations. The output of the system call can be stored in a trace (e.g., EIO trace in SimpleScalar) so that future simulations can use those traces instead of emulating the system call again. Using system call traces like EIO traces ensures deterministic simulation, and we describe this approach in more detail in Section III.B.

The above approach is not desirable for a number of reasons. First, the programmer writing the emulator needs to explicitly handle each system call to find the registers and memory locations that contain the input/output operands. This code is then only valid for a given operating system. To use the simulator on multiple operating systems would require the emulation of the simulated system calls for each of these systems. Even maintaining the simulator to run on the same operating system requires changes over time to support newer versions of

the operating system. Similarly, if the user desires to run a workload compiled for different versions of an operating system, the emulation may need to change if the operating system interface has changed. To top it all, complex system interactions due to asynchronous interrupts and DMA transfers cannot be handled easily with this form of emulation, which is required to correctly execute real world desktop applications like `acroread` and `powerpoint`.

This chapter described a simple binary instrumentation solution to capture the effects of all types of system interactions without having to explicitly emulate each system call or interrupt. Since our solution is independent of the operating system, it is very easy to provide simulator support to execute binaries compiled for various operating systems, as well as to allow the simulator to be compiled and executed in any operating system.

III.F.2 Full system simulation

There exist full system functional simulators like Simics [43], SimOS [59] and SoftSDV [70] that can emulate the full system including the operating system and all interaction with the external devices. Therefore, one option for building performance simulators would be to execute the binary inside a functional full system simulator and use that as a front end to feed traces of instructions executed to the cycle accurate performance simulator [28, 44, 18].

However, building and maintaining full system simulators is very expensive. It requires multiple person-years of effort to develop them. Also, they need to be modified constantly to support newer systems. In addition, the execution environment required for running real applications on full system simulators can be hard to reproduce, because of dependencies on specific kernel or device drivers versions, run-time license checking, elaborate installation procedures and large storage requirements. Therefore having a full system simulator in the front-end

incurs higher runtime overhead during simulation. Moreover, if the goal is to analyze the performance of just the user code then it is an unwarranted complexity to have a full system simulator as a front end.

It is highly desirable to have a way of handling all forms of system effects to correctly execute the application during simulation, but still preserve the simplicity of user-level simulators. The solution in this chapter is targeted toward achieving this goal.

III.F.3 Checkpoint Mechanisms

Detailed cycle accurate simulation of the full program execution is very time consuming. Sampling techniques like SimPoint [62] and SMARTS [76] are used to find representative samples of program execution. Simulating only these samples have been shown to provide accurate simulation results. The *Sample Starting Image* (SSI) is the state needed to accurately emulate and simulate the sample's execution to achieve the correct output for that sample. Various checkpoint mechanisms have been proposed to capture the SSI [58, 14, 66] with minimal checkpoint size. In this section these checkpoint mechanisms are described as they are related to the technique used to collect the pinSEL logs to capture system effects.

Szwed *et.al.* [66] proposed SimSnap, which instruments the application's binary, with the SSI corresponding to a sample and necessary code to restore it. Thus, during simulation, the simulated application's binary can itself restore the SSI for the sample to be simulated. To create such a binary, they first obtain the SSI of the application's state at the beginning of a sample by natively executing the instrumented binary of the application.

Ringenberg *et al.* [58] proposed an *Intrinsic Checkpointing* mechanism which also embeds SSI into the binaries and lets the application restore itself

during simulation. Their focus is to create one binary, that restores the SSI for all of the simulation points needed to simulate the execution of that binary, for a specific input. In doing this, they make an observation that to create the SSI for a simulation point they can take the ending memory image of the last simulation point, and just update it with all of the stores that occurred between the end of the last simulation point and the start of the current simulation point. In addition, they optimize the restoration process by choosing to restore only those locations that are read at least once inside the simulation interval. The intrinsic checkpointing approach achieves the purpose of checkpointing the SSI at the beginning of a simulation interval by having a list of memory stores that need to be executed to get the memory image up to date for the start of the new simulation interval. This saves a significant amount of space over storing the full memory image state for each simulation point. Note that the simulator using this binary with intrinsic checkpoints still needs to have support for emulating the system call and other system interactions. This is because the only thing that the intrinsic checkpoint scheme ensures is that the simulation point has the correct SSI. Thus, intrinsic checkpointing does not address the problems of handling system effects, which is the focus of the pinSEL approach.

Van Biesbrouck et.al. [14] also proposed an algorithm to reduce the size of SSI. Their technique assumes the EIO trace generation mechanism used in SimpleScalar to handle system calls. In the EIO traces generated by the default SimpleScalar, the SSI is the full memory image of the application at the beginning of the simulation interval along with a trace of result values of all the system calls executed (EIO trace) within the simulation interval. Instead of having the full memory image for SSI, they log initial memory values only for the locations that are accessed within the simulation interval. They also consider representing the same information in a different format in the form of Load Value Sequence (LVS)

which is essentially a trace of all the load instructions. Their approach focuses on reducing the size of the SSI, and not upon providing system call logging. They still rely upon the EIO traces and system call emulation in SimpleScalar for that. pinSEL's logging focus is to not have to provide any system emulation for SimpleScalar, while at the same time enabling the simulation of real (non SPEC) programs on SimpleScalar.

Checkpoints for Replay-Debugging

BugNet [50] is a hardware mechanism to record checkpoints during execution. When a program crashes, the checkpoints are used for replay debugging. BugNet also records user-level memory values in order to replay the execution deterministically. In their mechanism, every load value is logged unless a bit in the cache says that the word being loaded has already been logged. Upon executing a system call or interrupt, BugNet logs all the registers and resets all the bits in the cache line. By reseting the bits, the load locations changed by the system call or interrupt are logged. BugNet relies on cache coherence mechanisms to reset the bits of words changed by DMA transfers. In contrast, pinSEL does not rely on any hardware mechanism or operating system support. pinSEL maintains a copy of user-level memory and compare the values of the copy with the values in memory. By doing that, it can detect all forms of memory changes by the operating system with a simple comparison. Also, since pinSEL is not tied to the hardware, it can maintain a large copy of user memory and hence minimize the number of log entries. Finally, BugNet does not log the code executed by the application, whereas pinSEL provides a mechanism to log the code and the system effects to it, which allows correct handling of dynamically loaded libraries.

III.G Summary

One of the primary requirements for an architectural performance simulator is the ability to handle interactions with the system through system calls, asynchronous interrupts and DMA transfers. Conventional solutions such as SimpleScalar and pinLIT provide system support by emulating system calls, and they do not provide support to deal with asynchronous interrupts nor DMA transfers.

This chapter presented an automated logging solution for capturing system effects. The system effects are captured without the knowledge of the semantics of any system interaction. This was accomplished using a binary instrumentation tool to gather system effect logs, which are then used to guide architecture simulation. This approach is very easy to implement and is easy to port to other operating systems and architectures. Previously, SimpleScalar was capable of emulating only 81 system calls, which essentially limited its use to SPEC workloads. But with the help of the pinSEL logging support, it is now capable of simulating any linux application. As a result, any application that spends significant amount of time in application code and user level shared libraries can be evaluated using SimpleScalar.

Acknowledgments

Chapter III contains material that appears in "Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation", Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn and Brad Calder, in *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. The dissertation author was a primary investigator and author of this paper.

The implementation of the tools presented in this Chapter resulted from an internship realized by the author with the VSSAD group at Intel Corporation, during the year of 2006, at Hudson, Massachusetts. The author is very thankful for the support provided by the members of the Pin [42] Team, developers of the binary instrumentation engine. Special thanks go to Harish Patil, Robert Cohn, Greg Lueck, Chi-Keung (CK) Luk and Mark Charney for the technical support.

IV

Deterministic Simulation for Multi-Threaded Workloads on Multi-Processors

As traditional micro-processor designs, based on a single cores, hit the wall in terms of instruction level parallelism and power consumption, multi-core designs rise as an alternative to improve both performance and energy consumption [29]. These processors can increase the throughput of applications composed by many tasks or processes (server applications for instance). In addition, by dividing programs' execution into many tasks that share data and execute in parallel, multi-core processors can significantly speed-up the execution of some applications. Hence, in order to fully exploit the performance potential of multi-core designs, applications need to be made parallel. As a result, multi-threaded benchmarks where many threads of execution share an address space are becoming increasingly popular. These applications can achieve significant speed-ups when running on processors with multiple cores. One example of such benchmarks is SpecOMP [9].

In addition to a very large instruction count, benchmarks for future multi-core processors face yet another simulation challenge: *non-determinism*. The non-determinism comes from the fact that threads access shared memory in different order during simulation of different architecture configurations. This happens because the relative progress of threads change. For example, the order in which locks are acquired by threads in one architecture configuration can be different across two runs. Also, the number of cycles and instructions spent spinning for a lock can be different. The changes in execution paths can result in an increased number of instructions spinning for locks or a change in the functionality of the application. For instance, some multi-threaded applications may allow access to unprotected data structures, which results in non-determinism during the execution. As a result, the execution paths across two executions are not guaranteed to be the same. If the variation in the execution paths is significant, two simulation runs cannot be compared directly, because the amount and type of work performed differs across executions. The problem is worsened when the operating system behavior is also modeled, since changes in the architecture configuration can result in interrupts arriving at different points in the execution, causing the OS to schedule threads differently across runs. This non-determinism problem has been pointed out in previous research [7, 41, 11, 35, 30].

In this chapter, we present a technique to guarantee reproducible behavior of multi-threaded programs, when simulated under different architecture configurations. The technique focuses on user-level simulation of multi-threaded programs. It ensures the same execution paths by removing the sources of non-determinism during execution. The pinSEL tool [49], presented in chapter III, guarantees deterministic simulation of single and multi-threaded programs on uni-processors. The approach presented in this chapter builds upon it and extends pinSEL to guarantee reproducible behavior for multi-core processors' simu-

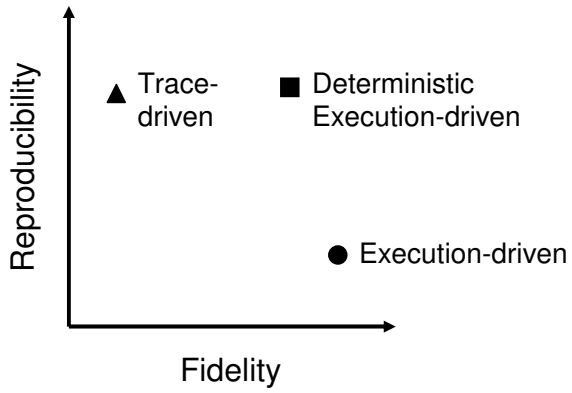

Figure IV.1: Comparison of deterministic execution-driven simulation with trace-driven and pure execution-driven simulation for multi-threaded workloads on multi-processors.

lations as well. The technique enforces the same order of shared memory accesses across simulations. The shared memory ordering is dictated by forcing threads to wait until dependencies are resolved, according to a pre-recorded order. Forcing threads to wait introduces artificial stalls, when running on a given architecture design. We present an approach to account and deal with these artificial stalls. The approach allows comparison across simulation runs to evaluate the best configuration. The stalls introduced are used as a measure of fidelity affected by the technique.

Our approach for deterministic simulation is execution-driven with the constraint that shared memory updates follow a fixed trace, which guarantees determinism. Figure IV.1 compares execution-driven deterministic simulation with trace-driven simulation and traditional execution-driven simulation. Deterministic execution-driven simulation is not as strict as trace-driven simulation because it allows non-shared memory operations to be interleaved according to the memory model used. It also allows simulation of wrong-path execution and hence a more accurate modeling of modern processor architectures. It provides

the same degree of reproducibility that trace-driven simulation does, because the execution paths are repeatable. Compared to pure execution-driven simulation, it does not provide a simulation with the same degree of fidelity because it forces the order of shared memory updates. Pure execution-driven simulation does not guarantee reproducibility though, which makes it difficult to compare results [7]. Although there is a loss in fidelity, sur deterministic execution-driven technique can provide error estimates, and give the designer confidence in the results.

The approach extends pinSEL to capture shared memory dependencies across threads. It logs the dependencies efficiently by using transitive optimizations. The technique reduces the number of dependencies logged while still allowing more relaxed memory models to be simulated. As in chapter III, the tool can capture checkpoints for full-program runs or only samples of execution.

The contributions presented in this chapter can be summarized as follows:

- An efficient mechanism to create simulation checkpoints of user-level code for large multi-threaded applications is presented. These checkpoints contain enough information to reproduce the execution path of the program exactly, on multi-core architectures. It allows efficient capturing of shared memory dependencies, with reasonable speeds and log sizes.
- The implementation of a deterministic simulator, which consumes the checkpoints, and provides 100% reproducible behavior (execution of the same control paths) across different architecture configurations, is described.
- Enforcing reproducibility during multi-threaded simulation introduces artificial stalls in the results. The techniques presented provide mechanisms to account for the artificial stalls, allowing comparison of two simulation runs for design space exploration.

IV.A Checkpoints for Reproducible Multi-Threaded Execution

The pinSEL checkpointing mechanism described in chapter III only guarantees reproducible simulation for multi-threaded programs running on uniprocessors. This section describes the extensions to handle reproducible simulation of multi-threaded programs on multi-processors.

In order to reproduce the execution of a multi-threaded execution on an uniprocessor, one needs to reproduce the exact thread context-switch interleavings as observed during logging. For uni-processor simulation pinSEL creates a *Context Switch Log*, which contains entries representing when threads should context switch. The Context Switch log is shared across all threads. In addition, the log files described in section III.C are created for each thread, so that each thread restores its own memory and registers during simulation. On multi-core processors however, recording the inter-leavings of threads as they are context switched is not sufficient to reproduce the execution. This is because threads are run in parallel and the execution depends on the order in which shared memory locations are updated. Hence an approach to record the shared memory dependencies across threads is needed.

IV.A.1 Logging Shared Memory Dependencies for Multi-Processors

This section explains the approach to record the shared memory dependencies across threads. For guaranteeing reproducible simulation, the order recorded needs to be obeyed during simulation. This means that the same exact execution paths and shared memory dependencies seen during logging, across the different threads, will be simulated from one simulation to the next.

In order to log shared memory dependencies, there are two sub-problems that need to be solved. The first problem is related to detecting these shared

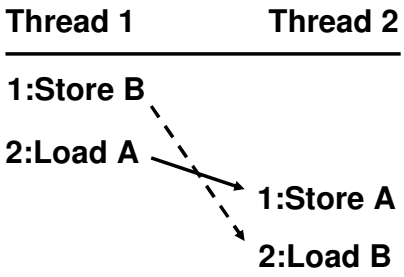

Figure IV.2: Netzer transitive optimization

memory dependencies during logging. The second problem is related to efficiently logging this information to reduce the log size. Previous hardware proposals [77, 50] observed that shared memory dependencies can be detected by just looking at the coherence messages in a multi-processor system. They used the Netzer transitive reduction algorithm to reduce the log size. The approach used for logging shared memory dependencies is similar to the hardware proposal [77], but it is implemented completely in software. It also implements the Netzer transitivity reduction algorithm [51] to minimize log sizes. The Netzer algorithm works by exploiting the transitive property in a system that assumes sequential consistency. For example in Figure IV.2, there is a read-after-write (RAW) dependency between $StoreB$ on thread 1 and $LoadB$ on thread 2. However, this dependency does not need to be logged, because the write-after-read (WAR) dependency between $LoadA$ on thread 1 and $StoreA$ on thread 2 transitively implies it. Later in the chapter, we present a discussion on how a sequentially consistent memory order is collected. This does not mean that it can only simulate sequentially consistent memory models, as explained in section IV.B.

For detecting shared memory dependencies, a global data structure that emulates a cache coherence directory is used. This data structure is a hash table indexed by the effective address of a memory operation. The table is referred to as a *directory* hereafter. Each entry represents a range of addresses and contains

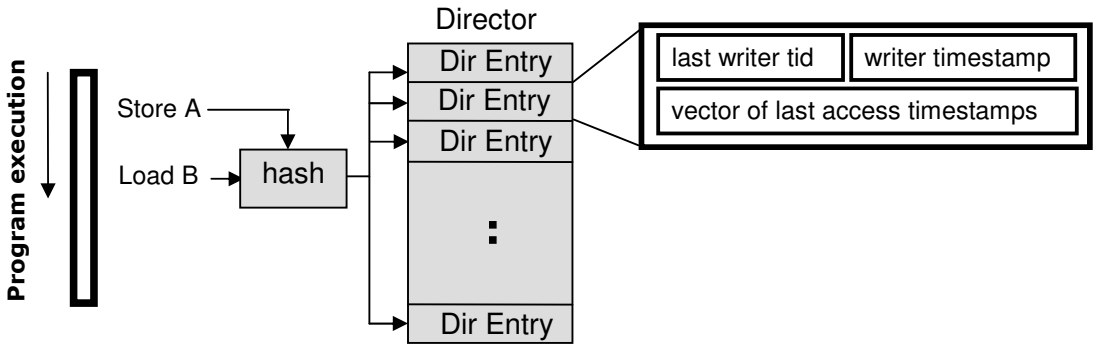

Figure IV.3: Directory table used to detect shared memory dependencies

the thread ID of the last thread to write to that address range, along with the timestamp of the memory operation that wrote to the address. The timestamps in this context are the *dynamic memory instruction count* since the beginning of the logging. In addition, each entry also contains a vector of the timestamps with an entry for each thread that accessed that address range. These timestamps are used to create a log entry that represents the dependency between two instructions across the threads. The timestamps are also used to implement the Netzer optimization. Figure IV.3 illustrates the table described above. The shared memory dependencies are logged in a *Race Log*, maintained per thread, which has entries in the following format:

```
local_mcount remote_tid remote_mcount
```

`local_mcount` is the memory count of the dependent thread, `remote_tid` is the thread ID of the remote thread upon which the local thread depends on, and `remote_mcount` is the memory count of the remote thread. The interpretation of a race log entry is as follows. The current thread, consuming the log, cannot execute its memory operation `local_mcount` until the remote thread identified by `remote_tid` executes its memory operation `remote_mcount`, because there is a dependency between the two. In order to minimize log sizes, only the difference

between current `local_mcount` and the previous one is logged. Similarly only the difference between the current `remote_mcount` and the previous one for the same thread is logged as well.

For every load and store instruction executed by the program, the directory entry where the instruction's effective address maps to is accessed. The entry is accessed to log shared memory dependencies and also to update its fields. The logging algorithm and the table update steps are explained now. The following actions happen for each instruction, depending on its type:

- **Load Instruction** - If the last writer ID for that thread is different than the current thread ID for the load, a read-after-write dependency is logged, if the dependency was not implied by another dependency already. The dependency is logged using the timestamp for the read instruction, the thread ID of the remote writer, and the timestamp of the write instruction. The vector of last access timestamps position for the thread accessing the directory entry is updated with the current memory count timestamp of that thread.

- **Store Instruction** - The vector of last accesses is checked and a dependency between every thread (except the thread executing the store) that last accessed that memory location is logged, if those dependencies were not implied by other dependencies already. Each dependency is logged using the timestamp for the store, the thread ID of each remote thread and the timestamp of that thread from the last access vector. Distinguishing between write-after-read and write-after-write dependencies is not needed (although it can be done by simply looking at the last writer ID and timestamp). Both the vector of last accesses and the last writer timestamp and ID are updated with the timestamp of the store operation and the thread's ID.

Figure IV.4 shows an instance of the directory table entries updated,

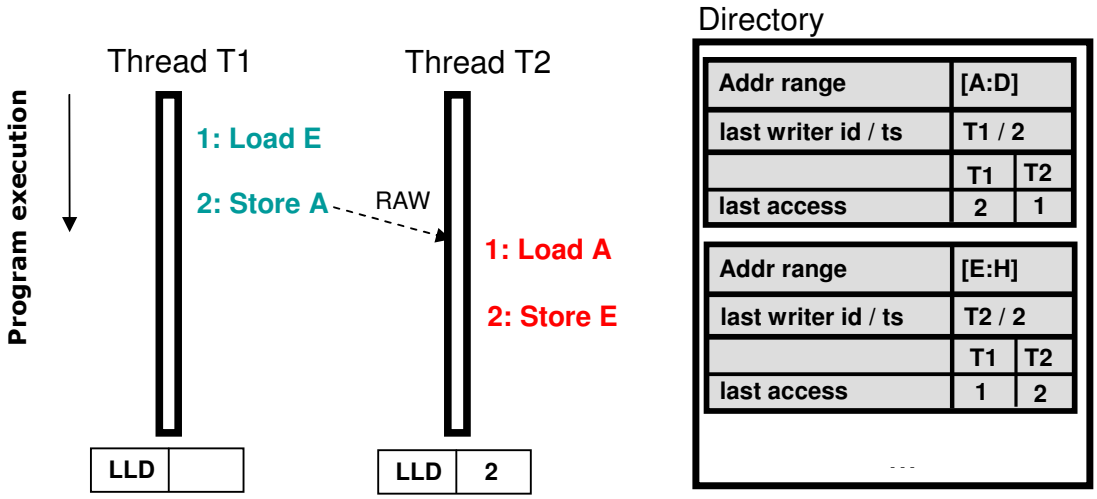

Figure IV.4: Example for the directory table state after hypothetical memory operations executed by threads 1 and 2. LLD stands for last-logged-dependency and *ts* stands for timestamp. Each directory entry shows the address range it represents, the last writer thread id and the timestamp when the write operation accessed the table, and the vector of last access from each thread to those locations.

after executing four instructions from thread $T1$ and $T2$. Each instruction is associated with a timestamp, which is the memory count for the thread executing it. The sequence of instructions shown in the figure represent the hypothetical order in which they executed (the total order is $(T1:1) \rightarrow (T1:2) \rightarrow (T2:1) \rightarrow (T2:2)$, where the tuples represent the thread id and the timestamp for the memory instruction). The table on the right hand side shows the state of the directory entries representing the address ranges $[A:D]$ and $[E:H]$. At bottom of each thread a vector of timestamps for each remote thread indicates the timestamp for the last dependency that was logged between the local thread and each remote thread (there is only one entry in the figure because there are only two threads in the example). This vector is called the last-logged-dependency vector (LLD). It is used to implement the Netzer transitive optimization. The figure shows the final state of the table. Let's now step through the execution of each instruction. Assume the table is initially empty. When *LoadE* executes, it updates the last access vector entry for thread $T1$ with the timestamp 1 in the directory entry corresponding to address range $[E:H]$. *StoreA* then executes and updates the last writer id and timestamp to $T1$ and 2 respectively, for the directory entry corresponding to address range $[A:D]$. No dependencies are logged at this point because the table was previously empty, and hence there are no dependencies. When *LoadA* executes, it looks up the directory entry $[A:D]$, and finds out that there was a last writer and the writer id is different than its own thread id. Hence a dependency is logged using the timestamp of *LoadA*, which is 1, and the thread id and timestamp of the last writer from the directory, which are $T1$ and 2 respectively. The last access vector position corresponding to thread $T2$ is also updated with *LoadA*'s timestamp. In addition, since a dependency was logged, the LLD entry corresponding to $T1$ is updated, indicating that a dependency with $T1$, using timestamp 2, was logged. Finally, *StoreE* executes.

It looks up the directory entry and finds out that $T1$ has accessed that address with timestamp 1 (for the *LoadE*). Hence a dependency needs to be logged. However, it finds out from the LLD that the last dependency logged with $T1$ was for timestamp 2. This means that the dependency is already implied, and therefore need not be logged. Even though no dependency is logged, *StoreE* updates the fields last writer id and last writer timestamp, with values $T2$ and 2, respectively.

Since the global hash table is shared across threads, each entry in the table is protected by a lock. This lock is the same lock used to guarantee instrumentation atomicity described in section III.C.7. This guarantees consistent state of the hash structure entries as well as gives a valid sequentially consistent order for the shared memory updates. This is because the implementation acquires and releases a lock before executing every memory operation in the instrumented application. As a result, when the tool sees that a memory operation to a shared location from thread A happened after a memory operation from thread B for the same location, it must be the same sequence observed by the processor.

By adding the *Race Logs*, the pinSEL checkpoints have enough information to reproduce the execution of multi-threaded workloads on a multi-core architecture. Next section presents a discussion about the changes introduced in the simulator to guarantee deterministic simulation for those workloads.

IV.A.2 Memory Model and Deterministic Simulation

A memory model determines the order in which reads and writes are allowed to execute in a multi-processor system. An execution-driven simulator allows the simulation of relaxed memory models because it can execute memory operations out-of-order, according to the timing model, as long as the restrictions imposed by the memory model are satisfied. The approach for deterministic

user-level simulation is execution-driven, but it is constrained by the race logs. Because it is execution-driven it also allows instructions to be executed in any order specified by the timing model, but only if no shared memory dependencies are violated. Hence it also allows simulation of relaxed memory models as well, but with the restriction that memory accesses to shared-memory obey a pre-determined order, which is in fact what provides the determinism. Given this restriction, using deterministic simulation for performance evaluation of different memory consistency models does not allow those dependencies to change during design exploration. Even though those dependencies are resolved in the order recorded, the technique proposed tracks when this occurs, and accounts for it when estimating the performance across two architecture configurations.

IV.A.3 Picking Samples for Simulation

The logging and simulation approach allows selecting the regions to checkpoint manually, or using techniques such as systematic sampling [76, 74] or Simpoint [62, 56]. For Simpoint, a two-pass approach is required, one for profiling the code and selecting the samples and another to generate the checkpoints. This two-pass approach can be implemented using the pinPLAY tool briefly described in section III.E. In that section, it was explained how the tool can be used to generate checkpoints with SimPoint, for single-threaded programs. The same technique could be applicable for multi-threaded programs. As noted in chapter II, a complete solution for picking simulation regions, using representative sampling, has not yet been provided. The contributions presented in this chapter take steps toward this direction. The focus of this chapter, though, is not on picking samples, but to show that the samples can be simulated deterministically for design space exploration and performance estimates can be provided. Therefore for the analysis provided in this chapter the samples are picked uniformly.

IV.B Deterministic Simulation

This section describes the changes made to an execution-driven simulator used by Intel Corporation, in order to consume our user-level checkpoints. This allows the simulator to reproduce the execution of workloads across different architecture configurations. This guarantees that the simulation is deterministic by ensuring the same execution paths for all threads, therefore making the executions comparable across simulation runs.

IV.B.1 Deterministic Simulation Implementation

The Asim [28] simulator was modified to consume the extended pinSEL checkpoints. Asim is a framework to create and maintain performance models. It allows for modular designs where components from different models can be put together to build another model, allowing Asim to be used for many performance models.

Asim separates the functional model from the performance model. The functional model is implemented as an instruction *feeder*. The performance model dictates the execution by asking the feeder to supply instructions. This allows a performance model to use different feeders and vice-versa, and simplifies the implementation of a new model, because the functional component is reused from previous implementations. A feeder which supplies instructions and memory values from pinSEL checkpoints was implemented. The feeder supplies register values and memory side effects for system calls by restoring those during simulation at the appropriate time from the checkpoints. In addition to that, the feeder dictates the order in which shared memory accesses are performed, to ensure determinism. By implementing a feeder, most processor performance models simulated by Asim can use our deterministic simulation approach.

Asim fits in the category of timing-first simulators [45]. This means

that the timing models controls which and when instructions should be executed. Through the implementation of well defined interfaces, the performance model controls the execution of the feeder by invoking method calls to fetch, decode, execute, perform memory operations, kill and commit an instruction. By using this interface, the simulation resembles the way the hardware executes instructions more closely. This implies that incorrect implementation of the timing model can lead to incorrect execution of the program, making it easier to identify bugs in the performance model. This also means that the feeder does not know that an instruction is speculative until it is notified by the timing model. Because of this, the functional model (feeder) has to implement support for rolling back instructions which are on the wrong path of execution. Different performance models call these methods at different point of the execution depending on the timing specifications of the model. This allows for great flexibility when re-using a given feeder. Figure IV.5 illustrates the interface between a performance model and a feeder. The arrows show how the control comes from the performance model. The implementation of the deterministic simulator adds feedback from the feeder to the performance model, which is used to control shared memory dependencies.

Enforcing Shared Memory Dependencies

The checkpoints for simulation of multi-threaded workloads contain a pre-determined order in which shared memory is accessed, which is the same order that was recorded during the collection of the logs.

During simulation the feeder needs to tell the performance model that certain instructions must wait until its shared memory dependencies with other threads are resolved. A dependency is resolved when the performance model makes the memory operation visible to the other processors in the model. For the models we use, where a *Processor Consistency* [5] memory model is imple-

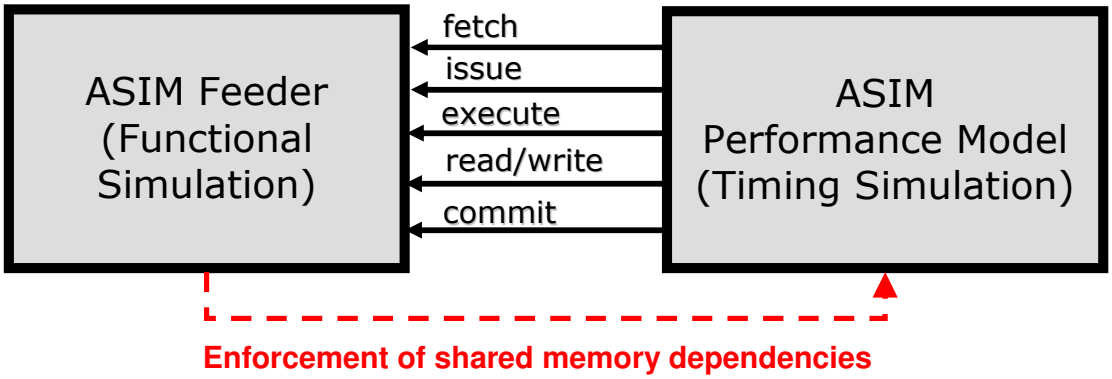

Figure IV.5: Deterministic simulation using Asim [28]. The feeder informs the performance model that certain instructions need to be synchronized. The feeder wakes up the instructions when the dependencies are satisfied.

mented, the reads are completed at commit time and the writes are visible when the memory interconnect network makes them available to other processors. At that point the write is completed. In the models where the deterministic simulation approach was implemented, when a memory operation is made visible to other processors, the feeder is notified, in order to update its own internal memory image. At this point, the feeder also notes whether shared memory dependencies have been satisfied or not. If they have been satisfied, the feeder notifies the performance model that those instructions no longer have shared-memory dependencies.

The pinSEL feeder knows the timestamps for all the instructions that have cross-thread dependencies. When fetching instructions, it checks if the timestamp of the instructions matches the timestamps of the next cross-thread dependency. If they match, the instruction is marked with a *cross-thread dependency* flag indicating that it cannot execute until the dependency is resolved. When an instruction is ready to be dispatched for execution, because all its operands dependencies are satisfied and the functional units are available, the simulator

checks whether this instructions has a logged cross-thread dependency, by checking the cross-thread dependency flag. If it does, the instruction is not allowed to dispatch until these dependencies are resolved as well. This incurs stalls cycles during the execution, which hereafter are referred to as *synchronization stalls*. These stalls would not naturally occur in the execution. Note that synchronization stalls can be also generated when executing a wrong-path. This is because the feeder does not know which instructions are speculative or not when fetching them and hence it will mark speculative instructions whose timestamps match the timestamp of shared memory dependencies. This is not a problem because the cycles spent synchronizing would be spent anyways executing the bad path, until it is killed.

Using Netzer Optimized Race Logs

In a cross-thread dependency, the *dependency source* is defined as the instruction in the remote thread and the *dependency destination* the instruction in the local thread. For instance, in a RAW dependency, the write is the source and the read the destination. In section IV.A it was mentioned that the Netzer optimization is used to log only the minimally necessary dependencies to enforce thread ordering in a sequentially consistent model. This reduces the size of the *Race Logs* significantly, by two orders of magnitude. Using the Netzer transitive optimization, though, requires that all the memory instructions before the dependency source must be completed before the dependency source instruction has completed its memory access. This is because in sequentially consistent processors, memory operations are completed in the program order.

Similarly, no memory instruction after the dependency destination instruction is allowed to execute before the destination instruction completes its memory operation. In Figure IV.2, *LoadA* is the source of the WAR dependency

and *StoreA* the destination. As a result, during simulation, no memory instructions after *StoreA* can execute before all memory instructions before *LoadA* have completed.

Ideally, a solution that can benefit from the reduction in number of dependencies, but still allow as much out-of-order execution of memory operations as the under-lying memory consistency model implemented allows is desired. This minimizes the amount of synchronization stalls when using deterministic simulation, hence increasing its fidelity. Our goal is then to be able to determine whether there are any potential conflicts between the memory operations "before the dependency source" and "after the dependency destination" instructions, without logging extra dependencies. This can be done by associating a *bloom filter* with each dependency logged. The bloom filter is a hash table indexed by the effective address of the memory instructions after the dependency destination. If the instruction has a dependency with any instruction before the dependency source, the bloom filter entry will have a one. Otherwise it will have a zero.

Since instructions must be committed in-order, the bloom filter only needs to contain the effective addresses of n instructions after the dependency destination, where n is the maximum number of instructions that can be in-flight during the execution. The effective addresses of these instructions need to be checked against the effective addresses of the instructions before the dependency source. An alternative to using the bloom filter would be to log all the dependencies without using netzer. This would result on two orders of magnitude increase in number of dependencies recorded in the *Race Log* file.

This bloom filter is built using a variation of the pinPLAY tool presented in section III.E. The tool consumes each checkpoint and executes the program under Pin. This tool is executed once to generate the bloom filters for each dependency, which is then added to the checkpoints. Since the tool executes

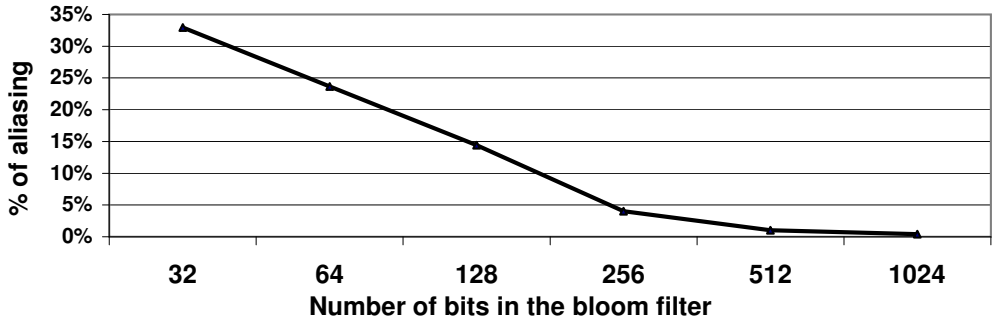

Figure IV.6: Percentage of instructions predicted as shared memory dependencies by the bloom filter due to aliasing as the number of bits used to implement it varies.

the instructions natively, because it uses binary translation, collecting the bloom filter for each checkpoint takes a very short amount of time, less than 10 seconds for $\pm$300 million instructions checkpoints.

During simulation, when a memory instruction is ready to dispatch past a dependency (an instruction which is younger than the waiting instruction) that has not been satisfied yet, it checks the bloom filter for that dependency. If the bloom filter tells it that it is safe to dispatch, it does not need to stall. Otherwise it has to stall because there is a potential dependency with an instruction before the dependency source that may not have executed yet.

Since a bloom filter is associated with each dependency, it is important to minimize its size to avoid an adverse effect on the log sizes. However, a bloom filter which is too small will result in too much aliasing, hence preventing instructions which do not have shared memory conflicts from being dispatched. In order to determine the size of the bloom filter to use, a study of the amount of aliasing resultant from the bloom filter as a function of its size was performed. This study was performed for the benchmarks used in section IV.E. Figure IV.6 shows the percentage of instructions that are predicted to have a shared memory conflict as the size of the bloom filter varies. A bloom filter of size 256 bits needs

32 bytes of storage. For the studies presented in section IV.E, a bloom filter size of 256 bits was used.

System Calls

The deterministic simulation approach presented in this chapter is targeted for user-level simulation. As a consequence, no operating system code is simulated while executing from the checkpoints. Instead the system call side effects are restored from our log files. This is not a problem with single-threaded programs. However, for multi-threaded programs, once system calls that executed during logging are not executed during simulation, the relative progress of threads with respect to one another is changed. Consequently a thread "jumps" ahead of the other threads during simulation, differently from what was observed when collecting the logs. To deal with this problem, whenever a thread executes a system call, pinSEL also logs the instruction count of all executing threads before and after the system call. The timestamps of all the other threads represent the state of those threads before and after the system call, and allows one to measure how much progress the other threads made while executing the system code.

Figure IV.7-(a) shows an example illustrating the mechanism. The figure shows the checkpointing run and the simulation run. On the checkpointing run thread 1 executes a system call (representing by the black portion of its execution). When the system call was invoked thread 2 was executing instructions X. After the execution of the system call, thread 2 was executing instruction Y.

During simulation, the system call instruction is not actually executed, it is just skipped. Its register and memory side effects are restored nonetheless, to guaranteed correct execution. To reproduce the exact behavior that was observed during logging, for that system call, the simulator forces the thread to synchronize with all the other threads before and after the system call, as shown

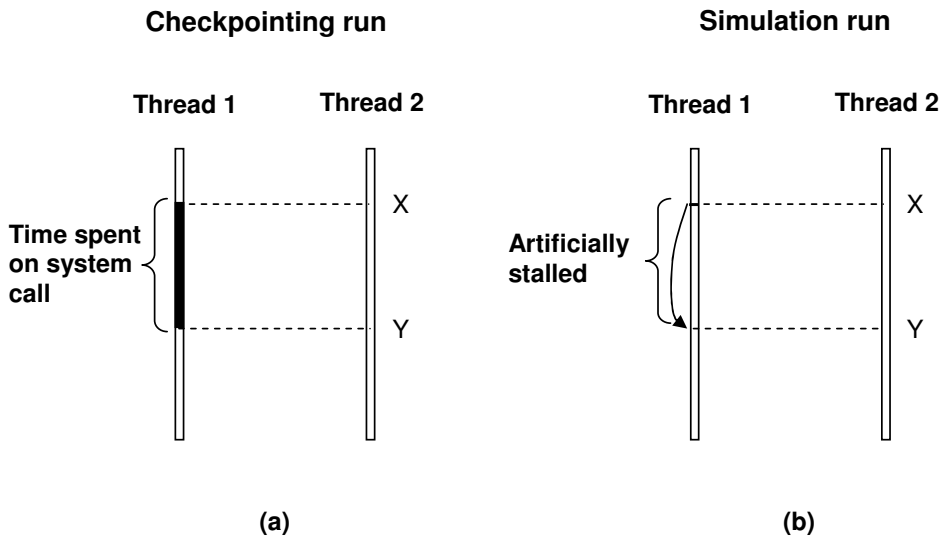

Figure IV.7: Problem with skipping system calls; (a) Checkpointing run; (b) Simulation run.

in Figure IV.7-(b). This allows it to model the time spent on the system calls during simulation, and to maintain the threads "in-sync" according to the logged execution. The stalls due to the system call are dealt with in the performance analysis described in the next section.

IV.C Comparing Samples across Architecture Configurations

The end goal of the deterministic simulation approach presented in this dissertation is to allow designers to evaluate the performance of a given architecture enhancement or feature. Hence throughout this section, it is assumed that a designer is comparing a *baseline* configuration with an *experimental* configuration. The designer is looking for an answer on how fast or how slow the experimental configuration is, relative to the baseline configuration, so he/she can decide whether it should be incorporated in a future processor design.

In the previous sections it was explained how to create the checkpoints for simulation and also how to enforce deterministic behavior during simulation. This guarantees that the same execution paths and therefore that the same amount and type of work is performed across both configurations, hence allowing one run in the baseline to be directly compared to another run in the experimental configuration. One consequence of enforcing reproducibility is that the simulator needs to introduce synchronization stalls that would not occur in the execution of the program. This section addresses how these stalls are taken into account, to calculate errors in the performance estimates provided by the technique. This involves comparing simulation samples to be able to determine which architecture configuration performs better. In section IV.E, we present quantitative results to show how the technique works for some design options experimented with.

IV.C.1 Differences Between Checkpointed Behavior and Baseline Configuration

When using the checkpoints, the program behavior simulated is for a valid execution of the benchmark on the machine where the checkpoints were collected. As a result, the relative progress of threads, when collecting the checkpoints, is likely to be different from the relative progress observed during the simulation. This is because these two machines are different. This difference results in synchronization stalls even for the baseline configuration. Later it is shown that many of these stalls are present in both the baseline and the experimental configuration, which means that they represent a bias in the same direction for both configurations, and therefore should not affect the relative performance comparison.

Another source of difference in behavior between the checkpoint creation run and the simulation comes from the fact that while collecting checkpoints for

the program, the binary instrumentation affects the behavior of the application by executing instrumentation instructions. Section IV.E shows how to deal with some of these effects originated from the instrumentation code.

IV.C.2 Classifying the Synchronization Stalls

As previously noted, the determinism comes at the cost of synchronization stalls added during the execution to keep the execution coherent with the checkpoints. The simulator keeps track of the synchronization stalls introduced during the simulation and divide them up in four categories:

- **True-Dependencies Stalls** - These are stalls needed to enforce the order of cross-thread RAW dependencies.
- **False-Dependencies Stalls** - These are stalls needed to enforce the order of cross-thread WAR/WAW dependencies.
- **Before-System-Call Stalls** - These stalls are introduced to make sure that whenever a thread is about to execute an instruction to invoke a system call, the other threads are approximately executing the same instructions as they were when the execution was recorded. This is needed to make sure the threads are in the same state as in the logged execution, when the system call is invoked.
- **After-System-Call Stalls** - These stalls are also introduced to model the time executing a system call. As explained in section IV.B.1, this is needed to maintain the threads in-sync with respect to the checkpoints, since it models the time spent executing the system calls in terms of instructions executed by the other threads.

The synchronization stalls are introduced to ensure determinism. These can translate to additional cycles spent during simulation when a thread is stalled

artificially, which needs to be tracked in order to determine an estimated error for the performance evaluation.

When the simulator is stalling an instruction due to a shared memory dependency, other instructions are allowed to be dispatched if: 1) they are older than the instruction waiting for the dependency to be satisfied; 2) they are younger but not a memory operation instruction; 3) they are younger and they are a memory instruction, but they have no potential cross-thread dependencies according to the bloom filter (associated with the youngest instruction stalling due to cross-thread dependencies), and they also have no operand dependencies pending. This means that while we are stalling an instruction due to a cross-thread dependency, other instructions can make progress. As a result, the pipeline is not completely stalled. Of course, if the synchronization stalls are long, eventually the pipeline will stall because internal processor queues (e.g. ROB) will fill up, preventing other instructions from making progress, or because the instruction is in the critical path of execution, causing other instructions to wait for it in order to dispatch.

In order to measure when an instruction that is stalled due to cross-thread dependencies is in the critical path, a technique similar to Tune *et al* [69] is used. The key observation is that an instruction is likely to be in the critical path if it reaches the bottom of the instruction queue before it is dispatched. In the deterministic simulator, whenever a thread is stalling due to a cross-thread dependency, the simulator keeps track of the number of cycles where instructions that are younger than the instruction stalling are allowed to dispatch. It also keeps track of the number of cycles where no instructions are not allowed to dispatch at all, and the oldest instruction in the queue is the instruction installing due to a cross-thread dependency. When the latter is true, this means that the pipeline is completely stalled due to the synchronization stalls. By doing this, it

can keep track of the artificial synchronization stalls during simulation that result in whole pipeline stalls. These are the stall cycles accounted for when calculating the performance estimates for the simulation.

IV.C.3 Matching Synchronization Stalls Across Configurations

When running a given checkpoint sample on two different configurations, the relative progress across threads can lead to different behavior of these threads on each configuration. During simulation, the difference in behavior will be translated in some threads reaching a shared memory update in different order than it happened in the original execution. This leads to a different number of synchronization stalls while running in each configuration. As suggested earlier, however, a number of these stalls are common across both configurations. This is because those stalls are present due to the intrinsic difference between the behavior captured and the behavior being simulated. The difference in behavior can introduce stalls in the simulation, but some of the stalls will be common for both configurations. The remaining stalls are a result of differences in the configurations. Those are a direct consequence of the variation in relative thread progress presented by the program as a result of the changes in the architecture. These need to be tracked precisely, so that a meaningful performance estimate for the simulation can be given. Consequently, the first step is to identify the common stalls across two runs, which will enable one to figure out the stalls resultant from configuration changes.

The mechanism to identify the stalls which are common across the configurations works as follows. First the two configurations are simulated. Each simulation run creates a file with all the dependencies which generated synchronization stalls. This file is referred to as a *stall trace*, and each entry in the file is called a *synchronization event*. A synchronization event is an instruction which

generates one of the stalls presented in section IV.C.2. For each synchronization event, a record with the thread ID and the instruction count for the instruction that generated the event is kept, along with the number of stalls generated. The thread ID and instruction counts uniquely identify the stall event. Because the behavior of the threads is deterministic across the simulation runs, these synchronization events can be identified across runs (by the thread ID and instruction count), if they generate stalls in both of them. Given that, the synchronization events across the runs can be matched up and the number of stalls which are common across them calculated. These are stalls originated from the same synchronization events. For example, if a dependency generated 50 synchronization stall cycles in one run, but 15 in the other, for the same event, 15 of those cycles are common. The other 35 cycles are only present in one run, due to differences in thread progress. The difference in stalls for each run is calculated and recorded in the stall traces. This difference is the total number of common synchronization stalls across all the runs subtracted from the total number of synchronization stalls for each run. This difference is later used to estimate the error when comparing simulations.

IV.C.4 Calculating Sample Speed-ups

Once the difference in synchronization stalls across the simulation runs is calculated, it is then used to estimate the error in performance resultant from the deterministic simulation. The error in performance estimate is due to artificially stalling threads to guarantee determinism. In this work, the metric to evaluate the performance gains across two samples is the weighed-speedup [67]. The formula to compute the speed-up is given in equation IV.1.

$$\text{ws} = \frac{1}{\#\text{threads}} \sum_{i \in \text{threads}} \frac{IPC_{\text{exp}_i}}{IPC_{\text{base}_i}} \tag{IV.1}$$

The formula equalizes the speed-ups on a per thread basis by dividing each thread IPC for the experimental configuration by its IPC in the baseline configuration. This metric is used because threads run at different rates of progress, when running on different configurations. As a result, different threads may reach the end of the sample in different runs. For example, assuming there are two threads $T1$ and $T2$. In one run, $T1$ runs faster than $T2$ and reaches the end of the sample before $T2$. At that point the sample is terminated because it is not fair to continue the simulation with only one thread running. This is because the thread only "terminated" executing the instructions in the sample, and continuing measuring the executing of only the other thread is not accurate, it is simply an artifact of using samples. On a different run though, thread $T2$ runs faster and finishes its instructions before $T1$. As a result, the instruction counts for each thread are different across the runs when simulating the samples. Using the weighted-speedup helps mitigate this effect, by equalizing the speed up across the threads.

The deterministic simulation can increase the number of simulated cycles as a result of the artificial synchronization stalls introduced. The increase in cycles is therefore a quantitative measure of error. Because of that these stalls need to be measured precisely, so the simulation can give the most accurate relative performance estimate for a sample, in the presence of the stalls. The error in performance estimation between two runs of deterministic simulation is proportional to the difference in synchronization stall cycles between the runs of the same sample in different configurations. The difference are the stalls not matched across the runs as explained in section IV.C.3. This allows one to calculate a range of IPCs that can expected from the run of the sample. One IPC includes the stalls introduced and not common across the runs, the other does not.

The method to compute the speed-up works as follows. First the simulations are run for both the baseline and the experiment configurations. Then the common stalls from the two runs are calculated, as described in section IV.C.3, on a per thread basis. The goal of this step is to find out the common stalls, which is the bias for both samples in the same direction. These do not affect the performance comparison. The non-common stalls in each configuration are used to compute two IPCs for each thread. One IPC including the stalls which are not common across the configurations, referred to as IPC^{STALLS}, and one not including those stalls, referred to as $IPC^{\text{NO-STALLS}}$. Figure IV.8-(a) illustrates it. It shows the hypothetical IPC for the baseline and the experiment. There are three IPCs shown in the figure for each run. The lowest IPC shows the IPC with all the synchronization stalls. This is before the stalls which are common across the configurations are factored out. Once the common stalls are subtracted, an IPC with the non-common synchronization stalls is obtained. This is what is called IPC^{STALLS}. The third IPC, the highest, includes no synchronization stalls at all. This is what is referred to as $IPC^{\text{NO-STALLS}}$. The difference between IPC^{STALLS} and $IPC^{\text{NO-STALLS}}$ is the range of IPCs expected from each configuration as a result of the stalls introduced due to the differences between the two configurations.

With both IPC^{STALLS} and $IPC^{\text{NO-STALLS}}$ for each thread, for the *baseline* and the *experimental configuration*, one can then compute two weighted speed-ups, ws_low and ws_high, as follows:

$$\text{ws\_low} = \frac{1}{\#\text{threads}} \sum_{i \in \text{threads}} \frac{IPC^{\text{STALLS}}_{\text{exp}_i}}{IPC^{\text{NO-STALLS}}_{\text{base}_i}} \tag{IV.2}$$

$$\text{ws\_high} = \frac{1}{\#\text{threads}} \sum_{i \in \text{threads}} \frac{IPC^{\text{NO-STALLS}}_{\text{exp}_i}}{IPC^{\text{STALLS}}_{\text{base}_i}} \tag{IV.3}$$

These two weighted speed-up calculations give a range of speed-ups

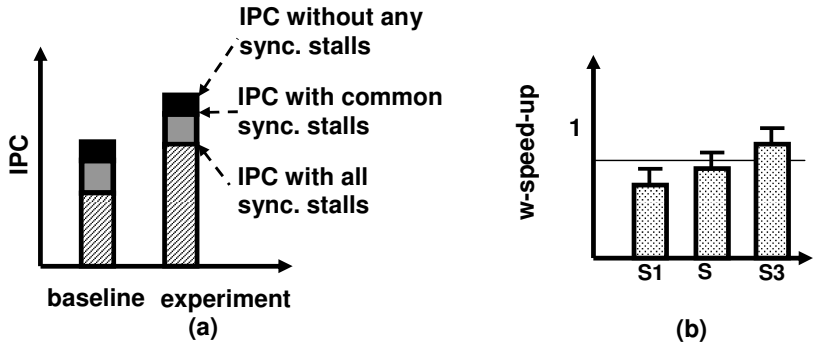

Figure IV.8: (a) - IPCs with all synchronization stalls, with only common stalls and without any stall; (b) - Weighted Speed-up Calculation

expected from the architectural experiment, considering the error, measured as the amount of synchronization stalls which are not common across the two runs. If this range is completely below or completely above 1, we can safely conclude that the *experiment* is either slower or faster than the *baseline* respectively. If the range of speed-ups contains 1 within its limits, then one cannot safely conclude if the experimental configuration is better than the baseline. These three situations are illustrated in Figure IV.8-(b), by the bars $S1, S2, S3$. For the first bar S1 we can safely conclude the experiment is slower than the baseline, the second bar S2 is inconclusive, and for the third bar S3, the experiment is faster than the baseline. From equations IV.2 and IV.3, it is also clear that the amount of stalls not common across the runs will determine the range of speed-ups, hence the need to compute it precisely. Section IV.E presents quantitative results on how the technique works for the benchmarks experimented with.

Table IV.1: Baseline simulator configuration

Core	2.4GHz, 4-issue, 128 ROB entries
Per Core Cache Hierarchy	Separate Instruction and Data caches 32KB, 8-way, 64-byte line size Unified second level: 256KB, 8-way, 64-byte line size

IV.D Methodology

To evaluate the proposed simulation methodology, we experimented with multi-threaded programs from the SpecOMP [9] benchmarks. Four threaded runs of these benchmarks were executed. The checkpoints were collected on a machine running with four Intel®Xeon™64bits CPUs operating at 3.66GHz, running Linux operating system.

For the results presented in section IV.E, the Asim [28] simulation framework was used, with a performance model simulating a hypothetical four core 64bits x86 processor. The baseline configuration relevant for this work is presented in Table IV.1. For this study, the number of threads is the same as the number of cores simulated. Hence each thread is mapped to one core. During simulation, a sample is terminated when the first thread in the sample executes all of its instructions for that sample.

We used the SpecOMP programs listed in table IV.2. These are parallel versions of spec programs. One of the advantages of use pinSEL is that it is based on binary instrumentation and hence execute faster (1 to 2 orders of magnitude) than functional simulators. As a result, we can collect samples for very large program runs. We collected samples for these benchmarks using their train and reference runs. The average instruction count for the programs in table IV.2 for

Table IV.2: SpecOMP programs used.

Program	**Description**
ammp	Computational Chemistry
applu	Parabolic/elliptic partial differential equations
apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants
equake	Finite element simulation, earthquake modeling
fma3d	Finite element crash simulation
galgel	Fluid dynamics: analysis of oscillatory instability
wupwise	Quantum chromodynamics

the reference input runs is 3.7 trillion instructions, and for the train runs 433 billion instructions. This work did not intend to choose representative samples for simulation, as discussed in section IV.A.3. In order to collect samples for testing the simulation methodology, the programs were executed under pinSEL and samples were collected uniformly throughout their executions. 10 samples were collected for each benchmark, each with approximately 300 million instructions. From these samples, we observed that the samples from the train input runs presented more time-varying behavior across samples. Hence we chose to use those samples for the experiments presented in the next section. For this programs the highest instuction count is for `wupwise`, with 1.5 trillion instructions and the lowest equake with 110 billion instructions.

Figure IV.10 shows the runtime overhead for collecting the logs, breaking it down in three components. The slowdown incurred by pin itself, without any instrumentation. The slowdown of a simple basic block profiler (`inst-prof`), which is the overhead of fast-forwarding between samples, and finally the additional overhead to collect the samples. The slowdown of Pin [42] by itself is quite low, in the order of 1.5x. The overhead for using `inst-prof`, compared to the native execution, is 9x. Using the pinSEL tool, this overhead increases to 31x. This

is because in addition to profiling the basic blocks, which is the only operation performed between the points where the checkpoints are collected, pinSEL also needs to create the checkpoints for each sample. In addition to the overhead of profiling the memory operations for logging the data, these runs also incur overheads for acquiring a lock before each memory operation and releasing it after, as well as logging shared memory dependencies. Figure IV.11 shows the checkpoint sizes to collect the logs. The average size of uncompressed SEL checkpoints is 119 megabytes per sample, which reduces to 35 megabytes when compressed with `bzip2`. The worst case log size, when compressed, is 138 megabytes per sample for `wupwise`. The best case is `galgel`, requiring only 3 megabytes per sample. The race log file, which stores the logged dependencies along with the bloom filters, represent about 6% of the log sizes. The size of the checkpoint log files is on average 43KB per one million memory read instructions.

The benchmarks were run for different configurations presented in section IV.E. After that the weighted speed-ups were computed, as presented in section IV.C.4, across the different runs. This gives a range of possible speed-ups that are discussed in the next section.

IV.E Evaluation

The technique for deterministic simulation was evaluated using the benchmarks mentioned in section IV.D. In this section, simulation results for experiments with different architectural configurations are described. The experiments change the cache size of the cores according table IV.3, and provide an estimate for the speed-ups achieved.

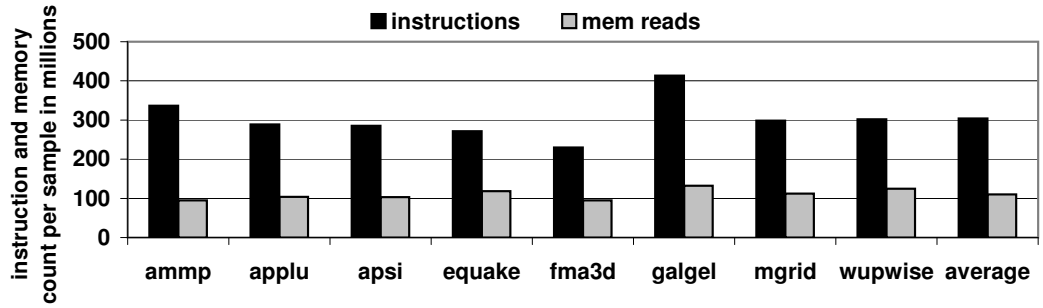

Figure IV.9: Average number of instructions and memory operations per sample for each benchmark

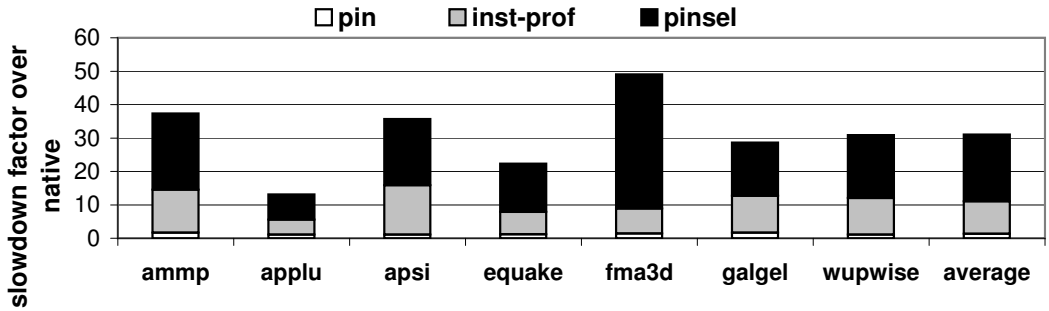

Figure IV.10: Slowdown to collect the 10 samples for each program

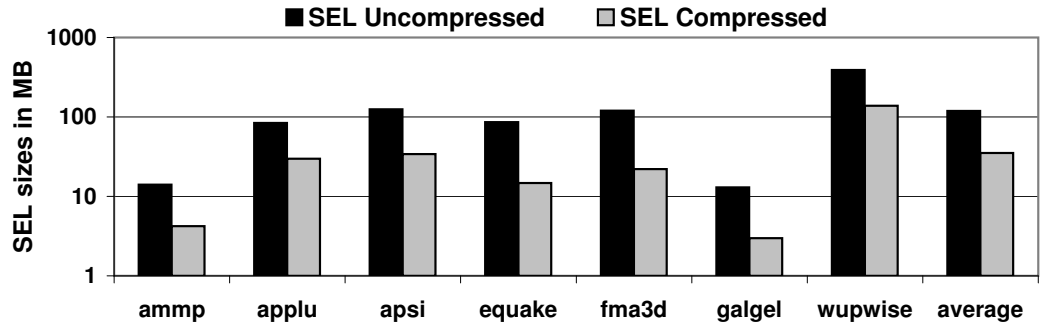

Figure IV.11: Log sizes of the SEL checkpoints per sample

Table IV.3: Experimental and baseline configurations.

Configuration name	**Parameters**
baseline	L1 Instruction and Data caches: 32KB, 8-way, 64-bytes line size L2 Unified cache: 256KB, 8-way, 64-bytes line size
cfg1	L1 Instruction and Data caches: 16KB, 8-way, 64-bytes line size L2 Unified cache: 128KB, 8-way, 64-bytes line size
cfg2	L1 Instruction and Data caches: 64KB, 8-way, 64-bytes line size L2 Unified cache: 512KB, 8-way, 64-bytes line size

IV.E.1 Estimating the speed-ups across simulation runs

The first set of results to look at is the number of synchronization stalls stalls present in the baseline configuration for the benchmarks we simulated. These are resultant from the differences in configuration from the machines were the traces were collected and also from alterations in program behavior originated from the instrumentation. Figure IV.12 shows the results for the SpecOMP benchmarks. The figure breaks down the synchronization stalls in the four categories we described in section IV.C. For these benchmarks, on average, 10.8% of the cycles spent during simulation of the samples are due synchronization stalls. From these, 6.5% of the cycles were due to system calls, and the 4.3% for shared memory synchronization stalls.

The synchronization stalls presented in figure IV.12 are intrinsic to the checkpoints, because of the reasons discussed in section IV.C.1. Those stalls are relative to simulating the checkpoints in the baseline configuration. In the next set of experiments simulation runs for two different configurations were performed. The objective was to examine how many synchronization stalls are common across configurations and, from that, determine the difference in stalls

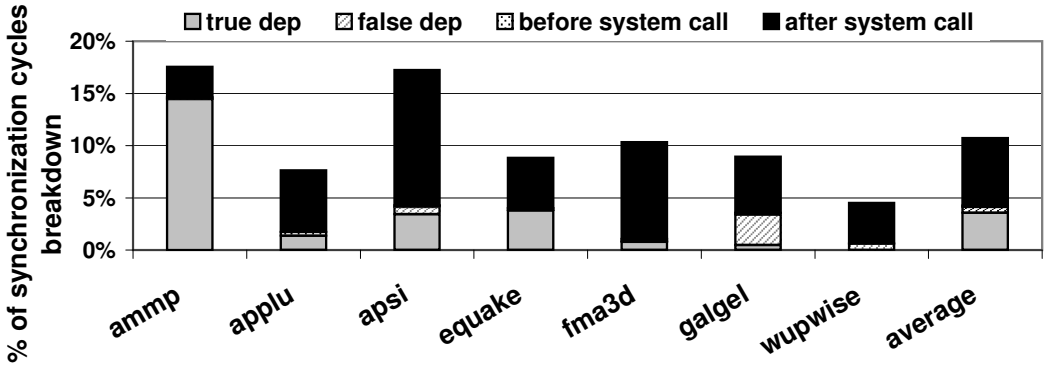

Figure IV.12: Percentage of synchronization stall for baseline configuration, broken down in categories: (a) true-dependencies (RAW); (b) false-dependencies (WAR/WAW); (c) Before-System-Call; (d) After-System-Call

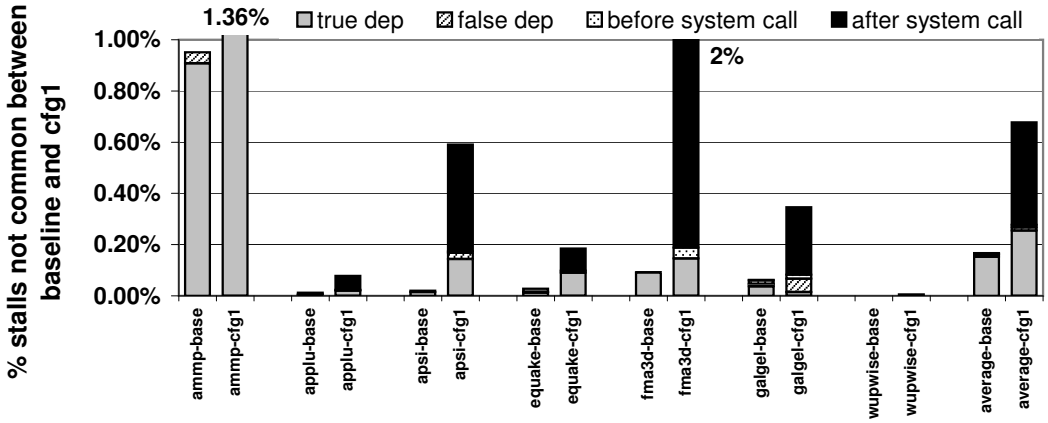

Figure IV.13: Percentage of synchronization stalls not common across the *baseline* and *cfg1*, w.r.t. the total number of cycles simulated

due to changes in the configurations simulated.

The configurations presented in Table IV.3 were simulated. Figure IV.13 shows the percentage of cycles relative to the total number of cycles simulated, which are not common across the runs of the *baseline* and *cfg1*. For `ammp`, for instance, just under 1% of the execution cycles for the baseline were spent with synchronization stalls which are not common with the synchronization stalls spent when running *cfg1*. Conversely, 1.36% of the cycles spent with synchronization stalls, when running *cfg1*, are not common with the baseline. `fma3d` had 2% of

its cycles, when running configuration *cfg1*, not common with the *baseline*. For the other SpecOMP programs, these percentages are smaller.

The differences in synchronization stalls presented in figure IV.13 correlate directly with the errors in speed-up predictions across the two configurations. This is because those differences are used to compute a range of IPCs for each thread and consequently the range of weighted speed-ups expected. Figure IV.14 shows the weighted speed-up computations between the *baseline* configuration and configurations *cfg1* and *cfg2*. The figure shows sets of four bars. The first two bars (*cfg1-nomatch* and *cfg2-nomatch*) show the weighted speedup results when using all the stalls in the computation (hence not using the approach to match the common stalls). This represents a scheme close to the approach proposed by prior work [41], where stalls are not matched across simulation runs. In their work, the error bars are not as high as shown in figure IV.13 because they collect their checkpoints using the simulator, while modeling the baseline configuration. Even so, during deterministic simulation, there can still be instrinsic stalls in the checkpoint as a result of the mechanism used to control the progress of threads. For example, in a RAW dependency, value of the write can be made visible one cycle before the read attempts to read it. However, during deterministic simulation, the read is stalled when its ready to dispatch. As a result, all the cycles between the ready to dispatch cycle and the cycle where the write is made available will be synchronization stalls, even though the checkpointing configuration is the same as the deterministic simulation run. As a result, considering the intrinsic checkpoint stalls result in higher error bars. Using the matching technique proposed in this chapter eliminates this problem. In our experiments, since the checkpoints are collected on a configuration different from the baseline, the intrinsic checkpoint stalls are higher, which results in higher error bars if no matching is performed. The remaining two bars in figure IV.13, labeled (*cfg1-*

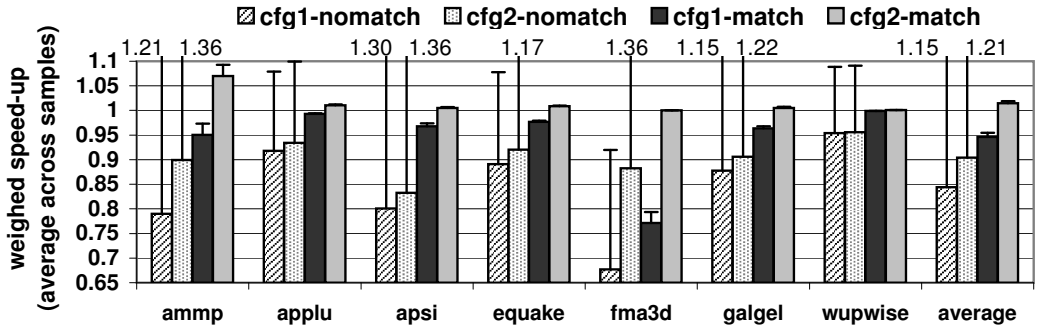

Figure IV.14: Weighted speed-ups computation for baseline against cfg1 and cfg2, when using only the non-common synchronization stalls across the runs, and when using all the synchronization stalls.

match and *cfg2-match*) show the weighted speed-ups when using the algorithm described in section IV.C.4, which matches the common stalls and uses only the difference in stalls to compute the estimates. `ammp` is the benchmark with the highest range of speed-up estimations. Between the *baseline* and *cfg1*, the speed-up ranges from 0.949 to 0.972. Between *baseline* and *cfg2*, the speed-up range is between 1.07 and 1.092. This difference is smaller for the other benchmarks, some of which are invisible in the figure. As expected, the error in the weighted speed-up calculation very closely tracks the percentages of synchronization stall cycles not common across the configurations. It is clear that not matching the stalls across the runs leads to very large speed-up range estimations, which do not give the designer the necessary confidence to make a decision. This emphasizes the importance of matching the common stalls across the configuration runs, one of the contributions of our work over the previous approach.

IV.E.2 Understanding the synchronization stalls

To better understand the nature of the synchronization stalls presented in figure IV.12, another set of measurements was realized. Some of the synchronization stalls observed are due to different behavior between the machine where

the checkpoints were collected and the simulated machine. Others are caused by alterations in the application behavior caused by the instrumentation code. To understand these better, the number of dependencies that generate synchronization stalls was gathered, classified by the length of those stalls. Figure IV.15 plots a histogram of the number of dependencies that generated stalls, separated by the length of the stalls. More than 90% of the dependencies that generate stalls generated stalls under one thousand cycles.

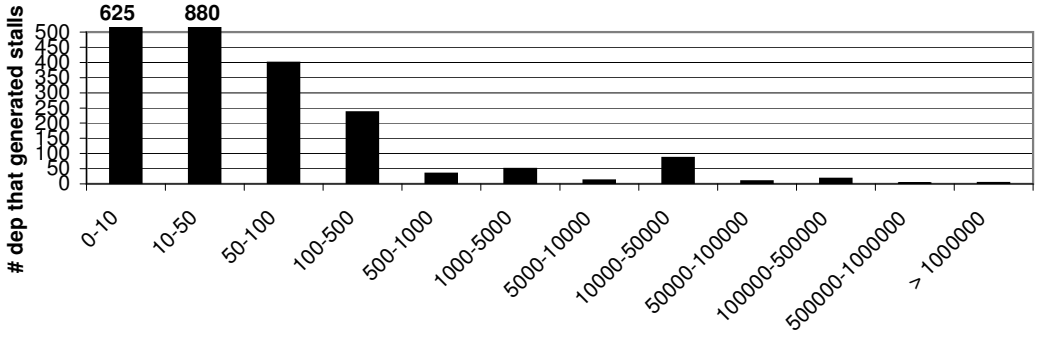

Figure IV.15: Histogram of number of dependencies that generate synchronization stalls, classified by stall length, across all programs.

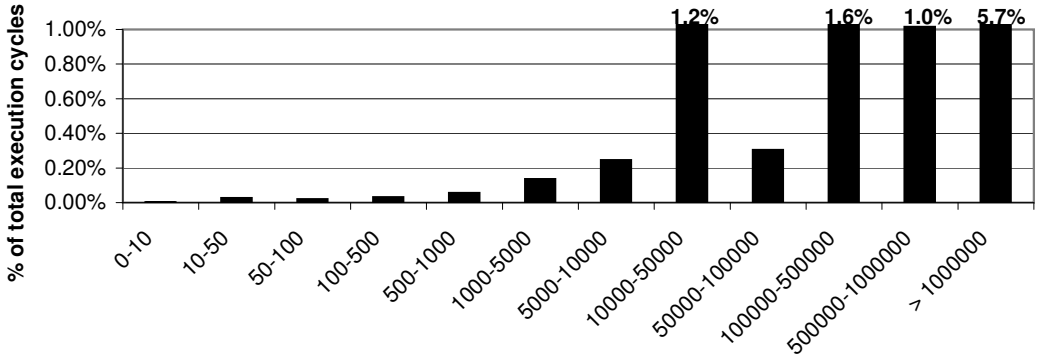

Figure IV.16: Histogram of percentage of synchronization stalls w.r.t to total number of cycles, classified by stall length

The next step was to find out what percentage of the synchronization stalls each of the stall ranges from figure IV.15 represent from the total number of synchronization stalls. This is shown in figure IV.16. From figure IV.12, one can see that about 10.8% of the total cycles executed on average are due to

synchronization stalls. So the percentages show on figure IV.16 add up to that same amount. The figure shows that a large number of the stalls are generated by a few dependencies. These dependencies stalls for a large number of cycles. This means that for those dependencies there is a large deviation in the behavior from the checkpoint and the behavior being simulated. Note that these stalls are for the baseline configuration runs, and not stalls resultant from simulating two different configurations. There are two reasons why this large stalls could happen:

1. The architecture configuration where the traces were collected is very different from the architecture configuration being simulated;
2. There are alterations in the behavior of the program because one of the analysis routines in the pinSEL tool took too long execute. This could happen because of a context switch while executing the analysis routine or just because the analysis routine spend a long time executing I/O operations, for example. These result in one thread not making progress and the other making progress, which causes long synchronization cycles during simulation.

Looking at the traces collected by pinSEL we observed that there are periods in the execution where one thread is not making progress while the other threads are executing instructions. This could happen because a thread is executing a system call. This case can be identified from the logs because the timestamps of all threads are logged before and after the system calls, in order to synchronize the thread before and after a system call. Hence long stalls due to system calls are properly identified and accounted as a system call stall. However, some of these long stalls observed are due to shared memory dependencies. The log files were examined to help understand why these stalls were being generated. By looking at the logs, we observed that many occurrences of the

long stalls due shared memory dependencies are for periods of execution where one thread is not making progress, and that is not happening due to a system call. These are conjectured to be stalls resultant from *heisenbug* effects caused by the instrumentation code. If these stalls are not due to heisenbug effects, they are resultant from a large difference between the machine where the checkpoints were collected and the simulated machine. These are due to a large deviation in behavior between the two machines.

Whether these large stalls are likely to be alterations in the behavior of the program's execution resultant from the instrumentation or a result of a large deviation between checkpointing machine and simulated machine, eliminating those effects from the samples is desirable. By doing this one can capture a more accurate behavior of the program, and at the same time reduce the amount of synchronization stalls resultant from these altered behaviors.

Once a simulation run is finished, the long stalls are eliminated as follows. The stall trace resultant from the simulation is examined. This trace will contain all the dependencies that originated stalls along with the number of stalls generated. The dependencies also contain the instruction counts of all threads at the beginning of the stall period and also at the end. The instruction counts allow one to find out the amount of progress other threads made while the dependent thread was synchronizing. This allows one to get a global view of the behavior of the simulation run during the synchronization stall. With this information we can then remove this slice of time from the simulation run, hence removing the altered or deviating behavior from the sample. This is effectively breaking down the sample in smaller samples, which do not include the slice in time when the long stalls occurred. Figure IV.17 shows an example. The synchronization stall period started when thread A needed to synchronize with some other thread. The synchronization period started when threads A, B and C had instructions

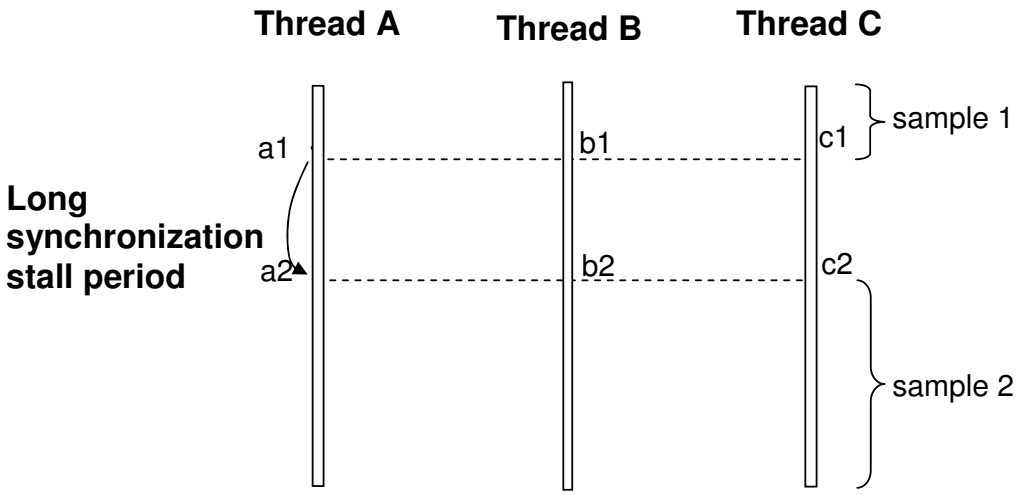

Figure IV.17: Sample breakdown representation. Long synchronization period starts at instruction counts $a1$, $b1$ and $c1$ and ends at instructions $a2$, $b2$, $c2$

with timestamps $a1$, $b1$ and $c1$ committed. It ended when when instructions with timestamps $a2$, $b2$ and $c2$ committed (if thread A is the one synchronizing, for instance, $a1$ and $a2$ will be consecutive numbers). Every synchronization stalls contains this information in the stall trace.

For our results, the samples were broken using synchronization stalls longer than $100,000$ cycles for the SpecOMP runs. The long synchronization is removed from sample, breaking it down into smaller samples. In the example from figure IV.17, the sample is broken into two smaller samples. The first starts at the beginning of the original sample and end at instructions counts $a1$, $b1$ and $c1$. The second starts at instruction counts $a2$, $b2$, and $c2$ and ends at the end of the original sample. The average size of the samples before breaking them down was ± 300 million instructions per sample, across all threads. After breaking them into smaller samples, the average size was reduced to 86 million instructions, with `ammp` the smallest average sample size of 1.6 million instructions, and `apsi` the largest, with an average sample of 114 million instructions. As the samples

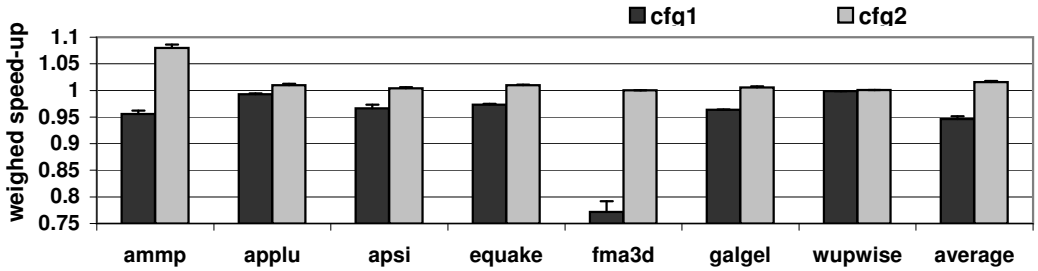

Figure IV.18: Weighted speed-up computation after breaking down the samples for eliminating the stalls longer than 100, 000 cycles from the baseline runs

become smaller, we need to add warm-up information into the checkpoints, to avoid an excessive simulation bias resultant from cold-start effects.

After the samples are broken, only remaining smaller samples are simulated with the configurations from Table IV.3. Figure IV.18 shows the speed-up estimation results when simulating these samples. The estimations are within the ranges presented in figure IV.14, but the error bars are much smaller, giving a more accurate estimation. All the programs except `fma3d` had their speed-up range estimations reduced significantly. `fma3d` still has many synchronization cycles under 100, 000, which are not common across the runs.

IV.E.3 Limitations of Deterministic Simulation

This chapter shows that deterministic simulation can be used for evaluating design changes for multi-threaded workloads. It has shown that for different cache configurations one can determine a range of speed-ups expected from the design change. One limitation of deterministic simulation is that order in which shared memory updates are performed is fixed across simulations. As a result deterministic is not as applicable when evaluating designs changes that require shared memory updates to be resolved differently. One example is comparing the performance of two different memory models. Nevertheless, deterministic simulation is useful for evaluating things such as cache configurations, branch

predictions and changes in pipeline width.

A second limitation of deterministic simulation is that for some design options, if the amount of synchronization stalls not common across the runs is too high, the results given by the simulation may not be conclusive. This happens when the range of speed-ups includes 1 within its limits, in which case one is not sure whether the experimental architectural is slower or faster than the baseline. If this happens, one can resort the method we presented to break down the samples when the amount of deviation is too high, in smaller samples. A alternative is to collect more samples to verify if the same results hold across all the samples.

IV.F Related Work

This section discusses prior work related to handling non-determinism when simulating multi-threaded workloads on multi-core architectures, simulating multi-threaded applications, and creating checkpoints for multi-threaded simulation.

IV.F.1 Dealing with Non-Determinism

Non-determinism in the execution of multi-threaded workloads has been recognized in prior research work [7, 41, 11, 35, 30]. Alameldeen *et al* [7] shows that for multi-threaded workloads, in particular server workloads, non-determinism can affect simulation results significantly, because the execution paths of the program and OS scheduling can change the behavior of the runs dramatically. If one run in each configuration is used to compare simulation results, there is a chance the conclusion reached is wrong because the behaviors of the runs was not the same. In some sense this is similar to comparing two runs of the same benchmark with different inputs. This is clearly inaccurate. They propose the use of statisti-

cal techniques to handle this problem. In their approach one simulates a program on the same configuration multiple times, inserting random perturbations to induce different behaviors (this is done because simulating the same benchmark on the same configuration is deterministic). This allow them to estimate the average behavior in that configuration within a confidence level. The same thing is done for a second configuration. For both configurations they then obtain the intervals of confidence for a given confidence level α. A confidence interval is a function of the number of samples (sample size in statistics terms) and the level of confidence desired. A sample here is a run of the program in one configuration. By increasing the number of samples and the level of confidence one can obtain a tighter interval, which is likely to contain the true estimated characteristic, e.g., the IPC of the run. Assume that experiments for configurations A and B are performed, and that it is know that A outperforms B (A's IPC is higher B's). Also assume that a confidence level $\alpha = 95\%$ is used to compute the confidence intervals. After running the experiments n times for each configuration one can obtain the intervals of confidence for each one of them. If these intervals do not overlap, there is a chance of at most 5% that the true IPC is not contained in the intervals. As a result, there is at most a 5% chance that the IPCs from each experiment may lead to the incorrect conclusion that B is better than A. If the confidence intervals do overlap, the probability of wrong conclusion would be higher. In that case, increasing n to get tighter intervals of confidence would be required. The cost of their technique is the requirement to run the program multiple times for the same configuration. Very small configuration changes can result in a large n, which can be impractical. The technique proposed in this chapter advocates running the program once for each configuration and comparing the runs directly. If there is variability across the runs, this variability will be translated into synchronization stalls cycles, which can then be used to obtain a

relative performance estimate. It also eliminates the sources of non-determinism due to OS scheduling because it focuses on user-level simulation.

Lepak *et al* [41] proposed deterministic simulation for full-system simulators. Their work presented the implementation of a full-system deterministic simulator, which also introduces artificial stalls to ensure determinism. The work presented in this chapter differs from theirs in many aspects. First it proposes a binary instrumentation approach for efficient collection of the checkpoints. This makes it practical to collect checkpoints for large applications such as SpecOMP programs. Second we propose an user-level deterministic simulation, which eliminates the non-deterministic behavior originated from the OS. To be able to use this for multi-threaded simulation, tracking synchronization stalls before and after OS calls needed to be tracked and included that in our error model. The most significant improvement made over the previous technique is the tracking and matching of the common stalls across simulation runs precisely, and only using the stalls that are different between the configurations towards the performance estimates. This significantly reduced the error bars over the prior results, and allows one to distinguish smaller speedups.

IV.G Summary

As the multi-core processor designs become mainstream, the use of multi-threaded applications to take full advantage of such designs is of primary importance. Simulating these benchmarks poses all the challenges that exist when simulating single-threaded programs. In addition, multi-threaded workloads suffer from non-determinism. This means that running the same benchmarks with the same inputs on different architecture configurations leads to a different execution paths across the runs. When the execution paths change, comparing the runs is no longer valid because the behaviors simulated are different.

This chapter presented a technique to handle the non-determinism problem in multi-threaded simulation for multi-core designs. The technique focuses on user-level deterministic simulation. The simulation is deterministic because the behavior of a workload is completely reproducible from run to run, by controlling the sources of non-determinism. An efficient technique to create checkpoints for deterministic simulation of multi-threaded workloads was presented. It contains a pre-determined order of shared memory updates, which are enforced during simulation. The chapter also presented the implementation of a deterministic simulator that consumes these checkpoints. The proposed technique introduces stalls during the simulation, which would not naturally occur during the execution of the program, so it can control the progress of threads and ensure a deterministic execution. A technique to account for and deal with these stalls in order to provide a performance estimate for the simulation runs was described.

Acknowledgements

Chapter IV contains material that appears in "Reproducible Simulation of Multi-Threaded Workloads for Architecture Design Exploration", Cristiano Pereira, Harish Patil and Brad Calder, submitted to the 14th International Symposium on High-Performance Computer Architecture, Salt Lake City, UT February 16-20, 2008. The dissertation author was the primary investigator and author of this paper.

The implementation of the tools presented in this Chapter resulted from an internship realized by the author with the VSSAD group at Intel Corporation, during the year of 2006, at Hudson, Massachusetts. The author is very thankful for the support provided by the members of the Pin [42] Team, developers of the binary instrumentation engine. Special thanks go to Harish Patil, Robert Cohn, Greg Lueck, Chi-Keung (CK) Luk and Mark Charney for their help and technical

support.

In addition, the modifications to the Asim simulator, also implemented while working for VSSAD, would not have been possible without the help of many in VSSAD and AMI groups. In particular, I would like to thank Brian Slechta, Chris Weaver, Joel Emer and Carl Beckmann for their help and technical support.

V

Summary and Future Challenges

Simulators are an indispensable tool for computer architecture research, both in industry and in academia. They are heavily used to evaluate current systems because of the level of visibility and flexibility they provide. By simulating current architectures, designers can understand and fix the bottlenecks for future processor generations. Simulators are even more important to characterizing the performance of future generation processors. Their use avoids the cost of building hardware prototypes, and provides the flexibility to experiment with many design options before implementation.

For a simulator to be useful, however, it needs to be accurate. Accuracy enables simulators to model complicated interactions among internal structures of the processor. A *de facto* standard in computer architecture simulation is the use of execution-driven cycle-accurate simulators. These simulators provide a high level of accuracy but at the cost of very long run-times due to implementation complexity. In addition to the complexity of simulators, another problem designers must address is the long running time of benchmarks, which can run for trillions of instructions. The standard solution to mitigate this problem is to choose representative samples from the execution of a program and only simulate these samples, which gives an accurate representation of the overall program

behavior. Techniques to do this efficiently were described in chapter II.

V.A Capturing operating system side effects automatically

Simulators can model the behavior of a machine in different levels of detail. Simulators can also model the entire machine, including the operating system, device drivers and user-level code, or only the user-level code. The type of simulator to use depends on the programs to be examined. While user-level simulators are simpler than full-system simulators, implementing user-level simulators is also a complicated task. A particularly difficult task is to emulate the operating system behavior so that programs can execute correctly. Operating system emulation is used in two contexts. One is to support the execution of programs during simulation when those programs interact with the system through system calls, asynchronous interrupts and DMA transfers. The other is to create checkpoints for user-level simulation. These checkpoints are used when performing sampled-based simulation and need to include the register state, code and data needed by the program during the simulation of the sample, as well the system interactions side effects to them.

Chapter III described in detail how state-of-the-art user-level simulators perform emulation of system interactions. The standard way to emulate system interactions, both in academia and in industry, is complex, tedious, difficult to maintain, and hard to port across different operating systems. It ties the simulator to the host architecture and may require re-implementing the entire emulation support when programs running on a different operating system, not supported by the simulator, need to be examined. This dissertation has presented a technique that trivializes operating system emulation in user-level simulators. The technique can be used to create checkpoints for full-program executions or for

simulation samples. The approach is based on binary instrumentation and uses an algorithm which is completely independent of the operating system and its intricacies. It not only provides support for discovering the side effects of system calls, but also of asynchronous interrupts and DMA transfers. The algorithm relies on binary instrumentation to maintain a shadow copy of user memory and to identify when memory values were changed by the system. Only the values changed by the system and used by the program are logged in a checkpoint. The checkpoint is then used to guide simulation. We implemented a tool called pinSEL, used to collect the results presented in the dissertation. The technique has been adopted by Intel Corporation to guide their simulators and has enabled Intel engineers to port their checkpointing tools to support simulation of programs from different operating systems (e.g. Mac OS and Windows), other than Linux. The only requirement is that the instrumentation tool (e.g. pinSEL) must run on the operating system for which the programs were compiled.

V.B Deterministic simulation of multi-threaded programs

More recently, with the widespread use of multi-core processors, simulation of multi-threaded programs in these processors is gaining special attention. These benchmarks present yet another challenge for designers to face when running experiments: the runs across different configurations are non-deterministic. The simulation runs can exercise different paths of execution because shared-memory locations are updated in different order. This makes the comparison across simulation runs difficult, because one does not know if the change in results is due to the hardware enhancement, or simply because the behavior of the programs has changed.

Chapter IV presented a technique to deal with the non-determinism problem. It first presents a technique to efficiently collect checkpoints for user-

level simulation of multi-threaded programs, running on multi-core systems. The technique extends the pinSEL tool to capture a log of shared memory dependencies that occur during the execution of the program. By capturing the order of shared memory updates and enforcing that order, runs of a multi-threaded program across different configurations are completely reproducible, such that the same execution paths will be repeated when running the programs on different hardware configurations. These checkpoints are used to guide the execution-driven simulation of multi-threaded programs on multi-core architectures. Using the checkpoints to guide simulation enables direct comparison of one simulation run in one configuration with another run in a different configuration. The determinism comes from the fact that threads are stalled to ensure that a predetermined order for shared memory accesses is obeyed. These stalls would not occur naturally during the execution of the program. For that reason, we also presented a technique to account for these stalls and to calculate an error measure in the performance estimates from simulations. The artificial stalls are used as an error measure to estimate the loss of fidelity in the simulation. It enables the calculation of a range of speed-ups expected from the experiments, which can then be used to make a design decision.

V.C Future Challenges

This dissertation introduced techniques that take steps toward simplifying and improving the ways designers build and use simulators. However, there are still challenges that need to be overcome to further improve the way simulators are used. In particular, one challenge concerns representative sampling for multi-threaded programs. The technique for deterministic simulation presented in this dissertation provides a solution to enable designers to compare samples simulated in different architectures. It does not provide a solution to pick repre-

sentative samples for simulation. Even though statistical sampling could be used to represent the execution of a program, the large number of checkpoint samples that need to be simulated can be an inconvenience. Furthermore, because the samples are usually small, warm-up issues have to be addressed very carefully. For these reasons, we believe that representative sampling, which picks a small number of larger, but very representative samples for simulation, is an attractive solution. In representative sampling, the program is profiled once, samples for simulation are chosen using machine learning techniques, and a checkpoint is generated for each sample. For representative sampling to work, a few challenges need to be overcome. First, the behavior that is profiled needs to be the same as the behavior checkpointed. Therefore an efficient technique to guarantee the same execution paths for the profiling and the checkpointing is needed. Efficiency is important because standard multi-threaded benchmark applications can be very long (trillions of instructions). Second, a technique to choose representative samples for multi-threaded programs must be studied. Perelman *et al* [56] proposed a scheme where samples are chosen from threads without considering the parallel behavior among them. The approach works well for programs in which each thread is executing the exact same code and the behavior of the threads are highly synchronized. If that is not the case, an approach that considers the parallel behavior of threads is needed. Such an approach would collect profiles considering the parallel behavior of the threads, instead of collecting profiles for each thread individually, and choose simulation samples from these profiles. When collecting profiles for each thread individually, the profiles are collected for intervals of execution specified as a number of dynamic instructions (e.g. 10 million instructions). For combined profiles, the profiles could be collected using intervals of execution considering a global dynamic instruction counts (e.g. 50 million instructions executed across all the threads). The samples would then be selected

using these combined profiles. Third, during simulation, one needs to guarantee that the same execution paths which were profiled and used to choose the sample as representative, need to be simulated, for better accuracy. This dissertation provides solutions that can be used to address the first and the third challenges described above. Improving the method for selecting samples on multi-threaded programs is one topic for future investigation. The end result would be the development of an entire methodology to: 1) profile the execution; 2) select the samples; 3) generate and simulate the checkpoints.

The results presented in chapter IV used checkpoints for four threaded benchmark runs. The checkpoints were collected on a four core machine and simulated on a four core target. However, tying the number of cores in the target model to the number of cores in the machine where the checkpoints are collected restricts the number of cores to be simulated. Therefore another topic for future investigation is characterization of the efficacy of our technique for deterministic simulation when the number of cores in the target is higher than the number of cores used to collect the checkpoints. This means that the number of application threads used when collecting the checkpoints is higher than the number of processor cores in the checkpointing machine. As the difference in the number of cores in the checkpointing machine and the number of cores in the target increases, the behavior of threads during simulation can potentially deviate more from the behavior of the checkpointing run, as this difference increases. In addition, as the number of cores in the target increases, there is a potential for more shared-memory dependencies across the threads. These could lead to more synchronization stalls using our approach. Hence a study to quantify these aspects and determine the utility of the approach is a natural next step.

Bibliography

[1] http://www.linux.org/.

[2] http://www.simplescalar.com/.

[3] http://www.spec.org/cpu/.

[4] http://www.spec.org/cpu2006/.

[5] IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, March, 2006.

[6] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.

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

[8] Arvind, K. Asanovic, C. Kozyrakis, S.-L. Lu, and M. Oskin. Ramp: Research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform, 2005.

[9] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, London, UK, 2001. Springer-Verlag.

[10] K. Barr, C. Weaver, T. Juan, and J. Emer. Simulating a chip multiprocessor with a symmetric multiprocessor. In *Boston Area Architecture Workshop*, 2005.

[11] K. C. Barr. *Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots*. PhD thesis, MIT, September 2006.

[12] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, 2006. ACM Press.

[13] R. Bhargava, J. Rubio, S. Kannan, L. K. John, D. Christie, and L. Klaes. Understanding the iimpact of x86/nt computing on microarchitecture. In *Chapter 10. Workload characterization of emerging computer applications. Kluwer Academic Publishers*, 2001.

[14] M. V. Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *International Conference on High Performance Embedded Architectures and Compilers*, Nov. 2005.

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

[16] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level checkpointing for shared memory programs. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 235–247, 2004.

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

[18] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti. Precise and accurate processor simulation. In *In Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads (CAECW)*, 2002.

[19] D. Chiou, H. Sanjeliwala, D. Sunwoo, Z. Xu, and N. Patil. Fpga-based fast, cycle-accurate, full-system simulators. In *Proceedings of the second Workshop on Architectural Research using FPGA Platforms (WARFP)*, February 2006.

[20] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *ICCD'96*, Oct. 1996.

[21] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *ICCD'96*, pages 468–477, Oct. 1996.

[22] T. T. P. P. Council. Tpc benchmark c: Standard specification. *http://www.tpc.org/tpcc/spec/tpcc current.pdf*, Dec 2003.

[23] J. Devore and R. Peck. *Statistics: The Exploration and Analysis of Data*. Brooks/Cole Publishing Company, 1997.

[24] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, 9(2), 2005.

[25] M. Dubois, F. Briggs, I. Patil, and M. Balagrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings 1986 International Conference on Parallel Processing*, pages 909–916, August 1986.

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

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

[28] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. 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.

[29] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[30] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 146–157, New York, NY, USA, 1993. ACM Press.

[31] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *ISPASS'03*, Mar. 2003.

[32] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[33] L. K. John and L. Eeckhout. *Performance Evaluation and Benchmarking*. CRC Press, 2005.

[34] A. J. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Comput. Archit. Lett.*, 1(1):7, 2006.

[35] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the validity of trace-driven simulation for multiprocessors. *SIGARCH Comput. Archit. News*, 19(3):244–253, 1991.

[36] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *WWC-3*, Sept. 2000.

[37] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS*, Mar. 2005.

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

[39] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical Report SMLI TR-93-22, Sun Microsystems Laboratories Inc., Dec. 1993.

[40] S. Lee, S. Das, V. Bertacco, T. Austin, D. Blaauw, and T. Mudge. Circuit-aware architectural simulation. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 305–310, New York, NY, USA, 2004. ACM Press.

[41] K. M. Lepak, H. W. Cain, and M. H. Lipasti. Redeeming ipc as a performance metric for multithreaded programs. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 232, Washington, DC, USA, 2003. IEEE Computer Society.

[42] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[43] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[44] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. *SIGMETRICS Perform. Eval. Rev.*, 30(1):108–116, 2002.

[45] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, New York, NY, USA, 2002. ACM Press.

[46] D. Model. Alpha axp workstation family performance brief - dec osf/1 axp.

[47] W. Mong and J. Zhu. Dynamosim: A trace-based dynamic compiled instruction set simulator. In *Proceedings of the International Conference on Computer Aided Design*, pages 131–136, November 2004.

[48] S. S. Mukherjee, S. K. Reinhardt, M. L. B. Falsafi, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin wind tunnel II: A fast and portable parallel architecture simulator. In *PAID'97*, June 1997.

[49] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 216–227, New York, NY, USA, 2006. ACM Press.

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

[51] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[52] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[53] 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*, Dec. 2004.

[54] M. Pellauer, J. Emer, and Arvind. Hasim: Implementing a partitioned performance model on an fpga, 2006.

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

[56] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *IEEE International Parallel and Distributed Processing Symposium*, pages 25–29, 2006.

[57] A. D. Pimentel and L. O. Hertzberger. Distributed simulation of multicomputer architectures with mermaid. In *SCS Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '98)*, July 1998.

[58] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In *ISPASS'05*, Mar. 2005.

[59] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[60] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2001.

[61] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT'01*, Sept. 2001.

[62] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, Oct. 2002.

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

[64] R. L. Sites. Alpha axp architecture. *Commun. ACM*, 36(2):33–44, 1993.

[65] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[66] P. Szwed, D. Marques, R. Buels, S. McKee, and M. Schulz. Simsnap: Fast-forwarding via native execution and application-level checkpointing. In *Proc. HPCA 2004 Interact-8: Workshop on the Interaction between Compilers and Computer Architectures*, Feb. 2004.

[67] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 318–327, Washington, DC, USA, 2001. IEEE Computer Society.

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

[69] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of the critical dependence path. In *Proceeedings of the 7th International Symposium On High Performance Computer Architecture*, 2001.

[70] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. Softsdv: A presilicon software development environment for the ia-64 architecture. In *Intel Technology Journal*, Dec. 1999.

[71] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170, 1997.

[72] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A practical fpga-based framework for novel cmp research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, New York, NY, USA, 2007. ACM Press.

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

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

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

[76] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA-30*, June 2003.

[77] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, 2003.

[78] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Third Annual Workshop on Modeling, Benchmarking and Simulation, held in conjunction with the 34th Annual International Symposium on Computer Architecture*, June 2007.

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

[80] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *HPCA-11*, Feb. 2005.