Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots

by

Kenneth C. Barr

B.S.E. Computer Engineering, University of Michigan (2000)
S.M., Massachusetts Institute of Technology (2002)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2006

Author .....
Department of Electrical Engineering and Computer Science
August 11, 2006

Certified by .....
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots

by

Kenneth C. Barr

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Computer architects rely heavily on software simulation to evaluate, refine, and validate new designs before they are implemented. However, simulation time continues to increase as computers become more complex and multicore designs become more common. This thesis investigates software structures and algorithms for quickly simulating modern cache-coherent multiprocessors by amortizing the time spent to simulate the memory system and branch predictors.

The Memory Timestamp Record (MTR) summarizes the directory and cache state of a multiprocessor system in a compact data structure. A single MTR snapshot is versatile enough to reconstruct the microarchitectural state resulting from various coherence protocols and cache organizations. The MTR may be quickly updated by each simulated processor during a fast-forwarding phase and optionally stored off-line for reuse.

To fill large branch prediction tables, we introduce Branch Predictor-based Compression (BPC) which compactly stores a branch trace so that it may be used to fill in any branch predictor structure. An entire BPC trace requires less space than single discrete predictor snapshots, and it may be decompressed 3–6× faster than performing functional simulation.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

Six years ago, Professor Krste Asanović invited a confused new graduate student into the SCALE group to port a simulator from Sun Solaris to x86/Linux. Since those nervous beginnings working from random Athena labs, I've had the opportunity to write Linux kernel drivers, debug hardware in the lab, study fine-grain leakage reduction techniques with Seongmoo Heo, and present the various ways of speeding up simulation that have led to this thesis. Along the way, the SCALE group has traveled to several countries and states where Krste is always willing to take us to dinner, introduce us to our peers in the field of computer architecture, and discuss research from every session of the conference. His ability to quickly "page in" for a 1-1 meeting and provide uncanny clarification and insight based on even my most muddled explanations never ceases to amaze me. I appreciate his accessibility, and I am grateful for his patience during Maslab and for the amount of respect and direction I've received over the past years. On top of all this, Krste works tirelessly behind the scenes to make sure our research is funded. In particular, the work appearing in this thesis was funded by the DARPA HPCS/IBM PERCS project, NSF Career Award CCR-0093354, and an Intel Ph.D. Fellowship that he urged me to apply for. Equipment was donated by the Intel Corporation. I thank Krste for securing this funding and the grantors for their vision, interest, and cooperation.

The inspiration for many of the ideas in this thesis came from a paper written by Roland Wunderlich, Thomas Wenisch, Babak Falsafi and James Hoe. Professor Hoe, a student of Professor Arvind, presented the idea during a visit to MIT. Excited about the talk, I mentioned it to Krste, and he put forth the idea of extending it to multiprocessors. I would like to thank Roland and Tom for their willingness to field questions about their research and for recognizing our work as related and not competitive.

I thank Professors Arvind and Larry Rudolph for agreeing to join my thesis committee and for providing comments on drafts of this work. I'm inspired by their distinguished careers, and I appreciate the insight that their experience has brought to my thesis.

Though I had technically met Joel Emer a few times before, we had our first official meeting after a lunch at Micro 36 in San Diego. There he told me about the ASIM simulation framework and made it possible for me to come on board as an intern in his group. He and Chris Weaver were generous of their time, helpful with my many questions, and patient as I slowly tracked down bugs in a newly parallelized simulator. Joel was an especially good mentor: he was willing to read lines of code with me and challenge me for full explanations when I resorted to hand-waving. I also would like to thank Joel for encouraging my thesis research direction once I returned to MIT, offering real-world insight, and suggesting the strategy that became BPC. Thanks to Andrew Beaumont-Smith and Brian Fisk for letting me tag along in their cars to Hudson. Thanks to Shubu Mukherjee for seeking interns at MIT and referring me to Chris and Joel.

The Open Source and Free Software movements have different philosophies, but have brought about similar results. Namely, it is possible for me to study and improve upon the source code to almost every application I use on a day-to-day basis. I use Firefox, Thunderbird, Pine, LaTeX, Ghostscript, Emacs, GCC, cvs, gzip, bzip2, rsync, openssh, SpamAssassin, Perl, Bochs, and more in a GNU/Linux environment. Not only do I appreciate that these tools are often more useful and flexible than their commercial counterparts, but I am also grateful that the authors of this software have released this code with licenses friendly to a graduate student like me.

I chose the SCALE Group not only because of its exciting research areas and excellent leader, but also because of its student members. They are friendly, helpful, funny, and smart, and have made my time at MIT fantastic. I'd especially like to thank Chris and Ronny — their breadth and depth of knowledge is matched only by their attention spans; and Mike and Heidi for their friendship and contributions to the MTR and its infrastructure. Thanks are due to Dave Wentzlaff of the RAW Group for his insight into computer architecture and MIT — not to mention encouraging me to take more bike rides.

Our research is supported by several staff members who go above and beyond the call of duty. For fending off hackers, upgrading computers at 6am, and catering to the whims of grad students, I thank our system administrator, Michael Vezza. Marilyn Pierce in the EECS Graduate Office cuts through red tape with aplomb, and Mary McDavitt keeps us well-fed and speedily reimbursed. Mary was especially helpful with thesis defense preparations and post-defense celebration.

Thanks for the proofreading assistance of Benson Barr, Naglaa Seifeldin, Caroline Dugopolski, and my committee. Chris Batten, Caroline Dugopolski, Mark Hampton, Ronny Krashinsky, Edwin Olson, and Rodric Rabbah helped me prepare for my oral defense. Their feedback was extremely helpful as I strived to introduce and unify the two major components of my thesis research.

The bulk of this thesis was presented as conference papers, and I am grateful for the input of the anonymous reviewers who guided me to important related work and, through their criticisms and suggestions, strengthened the ideas and refined the context of this thesis.

My family cherishes education in all of its forms. During my K-12 years, my parents were highly involved in my classroom-based learning. But, education extended outside the classroom. Museum visits, pinewood derbies, piano lessons, and trips around the United States all helped foster the level of curiosity that attracted me to graduate school and helped me work on projects like this thesis. Even though I'm now relatively far from home, I can count on my parents being interested in my work and my New England surroundings. Thank you Mom and Dad for everything you've done for me since I've arrived in Cambridge and long before. Thank you Michael for your capability to actually understand what I do. Thanks also to my grandparents, uncles, aunts, and cousins who have also shown enthusiastic interest in my education. Finally, I thank Carrie for her patience, enthusiasm, smiles, and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Imagine trying to quickly guess the race-day finish time of a marathon runner on a course he has never run. The most straightforward way would be to time him during a practice run of the entire 26.2 mile course. This practice time would likely be an accurate preview of race-day performance, but would require several hours to obtain. Perhaps there is a shortcut to estimating a runner's speed. One could spend a few minutes to drive the runner from the starting line to a tough hill at mile #17, time him as he ran 100 yards, and multiply that time by a scaling factor. Instead of waiting hours to measure a practice race, this technique provides an estimate with just a few minutes of driving and about 10 seconds of running. However, it is unlikely to be an accurate guess of the time it would take to run the full course. Such extrapolation fails for a number of reasons. First, the runner's performance is not only based on what lies ahead, but also on his current physical state. If he were really running the marathon, he would be quite tired at mile #17 and unlikely to run the 100 yards as fast as he did when he was not fatigued. Second, this particular 100 yards at mile #17 may be a steep hill on what otherwise was a flat course; the extra effort needed to climb the hill would not be needed the majority of the time, and one could end up overestimating the total course time. One could just as easily underestimate the time if the 100 yard segment was an easy downhill.

A program running on a computer can be thought of as a marathon. Segments of the race course correspond to phases of a program — steep climbs represent periods of low parallelism, while coasting down a hill is like executing a tight, carefully scheduled loop from the cache. The CPU, with its caches and branch predictors, corresponds to the runner and must be warmed-up before producing representative results. At the risk of belaboring the analogy, the multiple processors in a parallel computer could be thought of as a group of runners. Producing an accurate estimation of finish time by choosing which runners to measure at which points of the course seems as difficult as estimating the performance of a multicore computer running just a portion of a parallel workload.

Despite the flaws and obvious difficulties observed while trying to take a shortcut to measuring a runner's speed, this form of extrapolation was practiced by computer architects

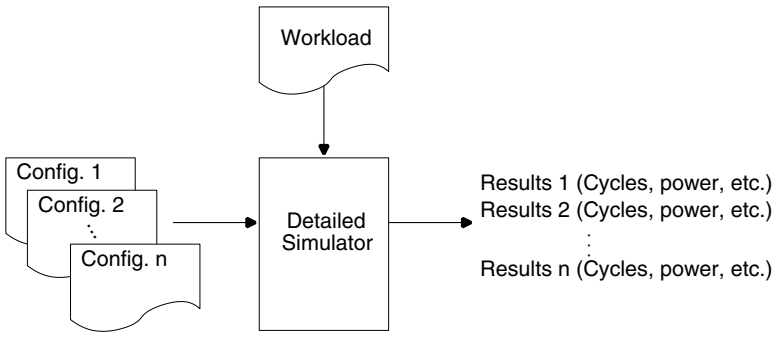


Figure 1-1. Using a detailed simulator to choose between several configurations.

not too long ago in an effort to reduce simulation time. Better simulation techniques have emerged for both uniprocessors and multiprocessors to strike a balance between accuracy and speed, but several obstacles remain. This thesis focuses on removing one of the obstacles: the inability to reuse simulation state over multiple experiments.

1.1 How computer architects use simulation

Computer architects rely heavily on simulators to evaluate, refine, and validate new designs before implementation. We use the term *host* to refer to the actual hardware running the simulation, and we use the term *target* to refer to the machine being simulated. Sometimes, especially in the context of virtual machines, the terms *host* and *guest* are used to denote the real and simulated computers respectively.

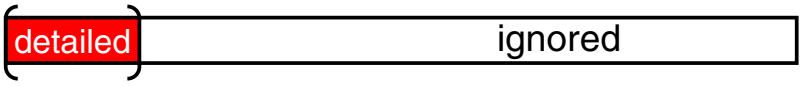
Typically, we seek to optimize a computer's performance on a workload (often called a benchmark) by adding features, updating the microarchitecture to incorporate research advances, or repairing performance bugs discovered in the field. Alternatively, we could be designing a radically different computer and wish to explore a large design space. To compare multiple configurations, we can perform several simulations as shown in Figure 1-1. A detailed simulator is configured to model a particular machine. For instance, Configuration 1 could specify a multiprocessor with small direct mapped caches in a 2D mesh, Configuration 2 could specify a multiprocessor with large associative caches in a ring, and so forth through Configuration n. The same workload is used with each configuration in separate experiments. The first experiment produces performance results for Configuration 1, and experiment n produces results for Configuration n. The results consist of statistics of interest to the architect such as cycles-per-instruction, power estimates, etc. The architect compares the results and determines which configuration is most desirable. Often the workload is a set of m applications rather than a single benchmark. In this case, a minimum of mn experiments are required: all n configurations are applied to each of the m applications. Other simulation methodologies may require additional experiments to capture the variation that can arise within a particular workload as we will later discuss in Chapter 5.

1.2 Motivation

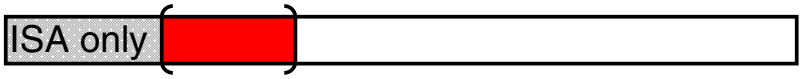
Ideally, one would simulate every relevant benchmark from beginning to end. But, due to the slowdown incurred by simulation (caused by bookkeeping, lack of host parallelism, software overhead, etc.), running complete benchmarks is prohibitive. For example, the most recent SPEC CINT2000 benchmark suite [106] contains 5.9 trillion instructions when run with reference inputs [55]. On modern hardware (a 3.06GHz Pentium 4), it requires about 31 minutes to complete the benchmark. However, on that same hardware, one of the fastest detailed, single-processor, out-of-order, superscalar models (SimpleScalar [14]) can only simulate about million instructions per second. At that rate, it would require over 72 days to complete one invocation of the SPEC CINT2000 suite. When additional detailed is added, such as cache-coherent memories and enough detail to boot the Linux kernel, simulation time becomes even more problematic. The full-system, multiprocessor, cache-coherent simulator we use in this thesis runs only 300,000 instructions per second, which translates to 228 days for the SPEC CINT2000 suite. Given this slowdown, it is no wonder that much work goes into finding accurate ways to speed up the simulation process.

To reduce simulation time, overall behavior can be estimated using short samples taken from a complete application run as seen in Figure 1-2. Many published architecture studies have chosen a single sample, either taken from the beginning (Figure 1-2(a)) or (when the program is known to begin with one-time initialization code) after some fixed number of instructions (Figure 1-2(b)). Such "fast-forwarding" is often performed with a functional simulator that simulates the instruction set architecture (ISA) only, updating only programmer-visible state such as the general purpose register file, control registers, and memory. The performance of modern microprocessors is greatly dependent on large quantities of microarchitectural state that is not visible to the programmer. Microarchitectural state, such as branch predictors and caches, must be initialized correctly at each sample point to avoid large systematic errors. To avoid cold-start effects due to fast-forwarding, microarchitectural structures can be warmed-up by a detailed simulator before detailed measurements are taken — a process called "detailed warming" (Figure 1-2(c)). Detailed warming greatly improves accuracy at the cost of increased simulation time.

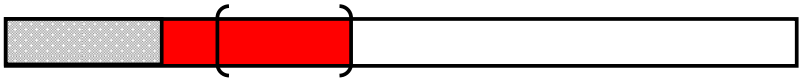
Applications generally contain multiple phases of execution with varying properties and much better characterization is possible by using multiple sample points spread throughout a run [26, 58]. Figure 1-2(d) depicts a metric, such as instructions-per-cycle, varying over the course of a benchmark. Three distinct phases are evident in the figure. Detailed measurements are performed in each phase, and the weighted combination of these measurements can be an accurate estimation of the overall performance. Various methods exist to identify program phases [57, 98, 100, 105] or to generate samples that are representative of a complete benchmark [33, 48].



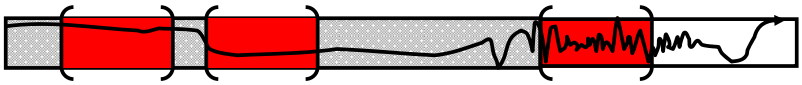
(a) Single sample



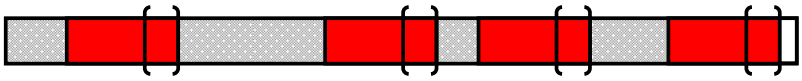
(b) Fast-forward + single sample



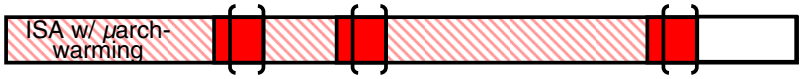
(c) Fast-forward + detailed warming + sample



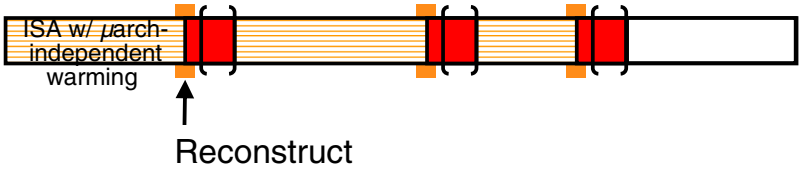
(d) Selective sampling



(e) Random sampling with lengthy detailed warming



(f) Random sampling with functional warming and short detailed warming



(g) Microarchitecture-Independent Snapshots (MINSnaps)

Figure 1-2. Various sampling and warming techniques. To minimize clutter, not all regions are labeled. Matching regions have matching color/shade. Brackets surround portion of sample that is measured.

Alternatively, many short samples can be gathered over the entire benchmark run. If these samples are independent, their average can be used to estimate the true mean of a performance metric, and the sample variance can be used to bound the estimation with a confidence interval (Figure 1-2(e)). This technique can be extended by performing *functional warming* during fast-forwarding [127]. With functional warming, large structures (caches and branch predictors) are kept warm with functionally correct models but without more costly timing simulation (Figure 1-2(f)). This reduces the amount of detailed warming required to get accurate results, as one need only to warm smaller short-lived structures. A downside of functional warming is that it warms only a *particular* cache and branch predictor. Other microarchitecture configurations require their own periods of functional warming.

Functional warming can be replaced with a period of fast updating of a microarchitecture-independent structure which can be used to reconstruct a variety of microarchitectures as shown in (Figure 1-2(g)). This **M**icroarchitecture-**IN**dependent **Snap**shot, which we call a MINSnap, stores the state of large structures; shorter structures may be warmed-up using a brief period of detailed warming. Chapter 3 shows that our MINSnap approach for memory system initialization can provide the same versatility as a detailed-warming approach in less time than a functional warming approach. Chapter 4 proposes and evaluates a technique to create a MINSnap for branch predictors. The application of MINSnaps to sample-based execution-driven simulation — and the construction of MINSnaps suitable for multiprocessor caches and branch predictors — is one of the main contributions of this thesis.

In all methodologies that require warming, the effort of fast-forwarding may be amortized by storing a checkpoint to disk that contains the state of the target prior to each sample point. The checkpoints can then be used to initialize a target microarchitecture configuration without repeating the fast-forwarding. MINSnaps provide additional time savings, as a single MINSnap may be used to initialize many target microarchitecture configurations.

1.3 Contributions of the thesis

Continued demands for faster simulation gave rise to the sampling techniques shown in Figure 1-2(a) through 1-2(f). The shortfalls of these techniques motivated the work of this thesis: the MINSnap approach shown in Figure 1-2(g). This thesis contributes new techniques for accelerating the software simulation of multiprocessors: initializing large on-chip structures using appropriate microarchitectural-independent snapshots (MINSnaps) and aggregating meaningful samples of execution. It introduces:

- **The Memory Timestamp Record (MTR):** a compact representation of cache and directory state from which various cache-coherent memory systems may be reconstructed.
- **Branch-Predictor based Compression (BPC):** a specialization of state-of-the art trace compressors that can be used to produce a microarchitecture-independent trace requiring less storage space than a collection of individual predictor checkpoints.

By combining these techniques, computer architects can quickly evaluate many design points in the multiprocessor design space without exorbitant storage requirements.

1.4 Thesis outline

Chapter 2 begins with an overview of modern computer architecture simulation. We describe the context of our simulation approach and explain the difference between prevailing styles of simulation. Next, we discuss sampling, focusing on how to advance to sample points, reduce cold-start effects, create checkpoints, and choose samples. We also discuss techniques for accelerating simulation that are complementary to our MINSnap approach.

Chapter 3 introduces and evaluates the MTR, and Chapter 4 presents the BPC. These two mechanisms can be combined with previous work in sampling to accelerate the simulation process. We evaluate the effectiveness of MTR and BPC using metrics of versatility, size, and speed.

Chapter 5 describes some challenges that remain in the field of simulating parallel targets. Specifically, it can be inaccurate to compare the outcome of two experiments unless one can be sure that the work being performed by the target is the same in both experiments. Another problem is the presence of variability that arises from simulating a complete computer system. Multiple experiments, each observing a different timing outcome, increase the time and/or number of hosts required by the architect. The cost of capturing variation further motivates the desire for faster simulation.

The thesis concludes in Chapter 6 with a discussion of remaining research questions and a summary of the work.

Chapter 2

Simulator Background

The popularity of various performance modeling techniques ebbs and flows as techniques are proposed, frameworks are distributed, architectures evolve, and a researcher's own hardware resources change. This chapter describes the recent history and state of the art in software performance modeling. Trace-driven simulation was popular through the 1990s. However, the intricate timing-dependent behavior of modern architectures cannot be accurately captured with simple trace-driven simulation, thus more time-intensive execution-driven simulation is typically used. Simulation time continues to increase as microarchitectures become more complex and multicore designs become more common. Though host machines grow faster, benchmarks grow longer and the complexity of execution-driven simulation prevents corresponding speedup. Today, researchers have rediscovered the sampling techniques once used to reduce the size of traces, and research in program phase identification aides in the selection of samples. The MTR and BPC, described in later chapters, are inspired by earlier related work. The MTR and BPC extend current sampling approaches to simulation, allowing fast simulation of multiple target configurations from a single stored snapshot or compressed trace.

2.1 Levels of detail

Simulation of computer architectures occurs along a wide spectrum of detail as shown in Table 2.1. At the lowest levels, one can model the physical behavior of transistors with *circuit level* simulation. This provides information such as the voltage and current at circuit nodes at a given time. *Gate level* simulation uses more abstract models of devices. Rather than individual transistors, it can model logic gates (NAND, OR, XOR, etc.) connected by wires. A gate level simulation tracks the state (0 or 1) of every wire in the system at every unit of time. A simulation written in a *Register Transfer Level (RTL)* language uses abstractions for most combinational logic such as comparators and adders, and it introduces registers to store state. RTL tracks every state bit at every clock cycle. For instance, we can express the incrementing of a program counter by writing an expression like `PC=PC+4`, rather than

Level of Detail
Analytical
Functional
High-level language, cycle-accurate
Register Transfer Level (RTL)
Gate
Circuit

Table 2.1. Levels of simulation detail.

modeling every gate of the register and adder. When RTL is too restrictive, architects can use *high-level language, cycle-accurate* models. Written in a language like C++, they allow for behavior to be described more abstractly than RTL models, reducing the effort needed to construct the model and increasing the ability to parameterize the model. The timing behavior of the system can be approximated without knowing the RTL implementation. A *functional* model must correctly simulate the functionality of the computer, but it makes no attempt to model its performance. At the highest levels, one can use an *analytical* model that, for instance, describes the computer as a queuing system so that it can be analyzed in terms of service times and queuing delay. In general, the precise models at the lowest levels provide the most detail at the expense of time. The higher the level, the more abstract and approximate the model becomes and the quicker it runs. In this thesis, we focus on functional and high-level, cycle-accurate models, and we discuss both in more detail below.

2.1.1 Functional simulation

Functional simulation is used in cases where it is sufficient to simulate only the programmer-visible state of the computer. A functional simulator simply decodes and executes every instruction in a program. At a minimum, it knows how these instructions manipulate the target's program counter, general-purpose registers, and memory. Such a simulator can execute a target program provided that it does not attempt to perform privileged instructions such as input/output routines. However, by simulating the functionality of system devices (e.g., disks, timers, and interrupt controllers), a functional simulator can run an operating system to handle I/O and other system calls. To reduce the effort required to build a simulator, it is common to trap system call instructions and use the host's facilities to perform I/O tasks such as reading and writing files. The tradeoff of using the host to perform system calls is that target activity is not measured during these operations.

Functional simulators are useful in many situations. In educational settings they may model a MIPS target on a ubiquitous x86 host to teach students about assembly programming using a relatively simple RISC ISA. Another common usage is to enable software bring-up on systems before they are released. Hardware manufacturers will often provide ISA simulators to independent software vendors before an actual computer is available. Programs

that run successfully in the simulated environment should run successfully when the real system is delivered. Functional simulators can help characterize some dynamic and/or input-dependent characteristics of a program including its instruction count, instruction mix, basic block distribution, branch characteristics, etc. The output of a functional simulator can be used as the input to a more detailed model to provide estimates of timing or power.

2.1.2 High-level, cycle-accurate simulation (Detailed Simulation)

We use the term *detailed simulation* as shorthand to describe cycle-accurate simulation implemented in a high-level language like C or C++. Detailed simulation is the process of using simulators to model the performance of a computer — not merely its functionality. The definition of "performance" is vague. Among other definitions, it can refer to the time required to run a given application program; a certain balance between speed, power, and silicon area; or the ability to meet power density constraints. Thus, even a detailed simulation can be divided into various sublevels of detail, and the term can best be defined as a simulation which informs the operator of those performance characteristics in which he is interested.

Performance, however it is defined, depends on microarchitectural structures that are hidden from the view of the programmer. For instance, to reduce the average time required to transfer a block of memory from a DRAM to a register, a cache hierarchy is created. Frequently accessed memory is stored in a small, fast cache. When the program reads or writes data that is found in the cache, it need not suffer the long latency of a DRAM access. Detailed simulators account for the varying memory access times caused by caching. They may also model the performance impact of features such as parallel functional units, out-of-order execution, shared buses, and predictors. A detailed simulator not only models the functionality of the computer, but also provides the architect with performance information such as timing estimates, power estimates, or more fine-grain statistics such as cache miss rates, activity counts, and contention information.

2.2 Detailed simulation styles

Putting aside functional simulation for a moment, we consider two styles of detailed simulation: *trace-driven simulation* processes a trace captured from real or simulated hardware, and *execution-driven simulation* allows the machine timing model to affect the workload's instructions and their order.

2.2.1 Trace-driven simulation

Trace-driven simulation is a technique in which a trace, or list of events, is recorded and analyzed. The events in a trace may include details of every instruction, memory access, and system event (e.g., interrupts) or merely the events of interest for a given experiment. The trace may be stored off-line for future analysis or consumed online by the simulator. One

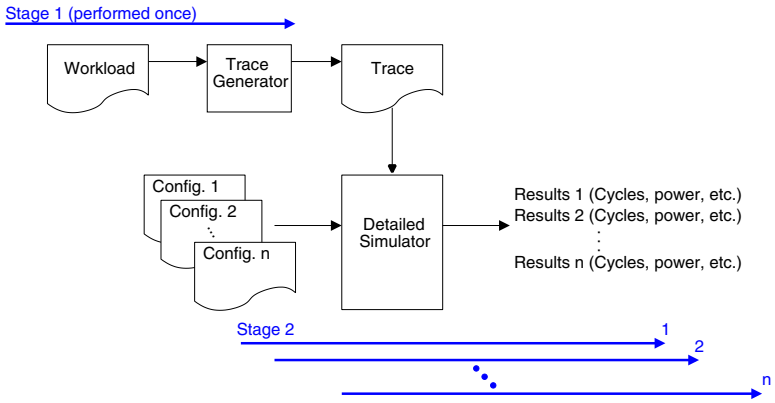


Figure 2-1. Trace-driven simulation

collects a trace by connecting a physical probe to a bus in a hardware implementation [22, 39], by modifying microcode [4], by adding instrumentation routines to an existing program [67, 104], or by simulating the program and capturing interesting events. Depending on the contents of a trace, it may be generic and useful across many platforms or tied to a particular ISA. Figure 2-1 shows the two stages of trace-driven simulation. First, a workload is fed into a trace generator. Once the trace has been generated, it may be reused over the course of n experiments, one for each configuration of the detailed simulator.

Traces have many useful properties including ease of collection, ease of use, portability, and determinism. Analysis of a stored trace may take place at a later time. While a trace can amortize the work of decoding and executing instructions across many experiments, a trace-driven methodology is subject to the speeds of the detailed simulator that processes the trace [118]. Trace-driven simulation enjoyed popularity through the mid-1990s. However, the rise of multiprocessors and speculative execution has limited the effectiveness of traditional trace-driven techniques. The difficulty with a trace is that it represents a particular set of instructions in a particular order. Modern computer systems can correctly execute a single program in many ways. The program may be interrupted by an operating system scheduler and interleaved with instructions from another process. Synchronization constructs may even alter a single process's instruction stream. A classic example is "busy waiting." While a process waits for a memory location to change, it performs a variable amount of loads and branches in a tight loop until it is allowed to proceed. The precise number of instructions can depend on microarchitectural parameters.

Elements of the microarchitecture may also cause events to occur that are difficult to record in a trace. These events and their timing relative to a program's instruction stream can vary as one refines the microarchitecture. For example, consider a classic five-stage pipeline that resolves branches during the third stage. If it predicts every branch to be not-taken, and resolves the branch to be taken, then the two instructions in the first two stages must be killed. Frequent mispredicts will significantly decrease the throughput of the pipeline. An architect may propose branch prediction to solve this problem, but it is difficult

to study the effects of branch prediction using a simple trace: when the predictor incorrectly guesses that a branch is taken to a certain address, one would like to fetch instructions from that address (which effects performance via the instruction cache). However, the trace contains only the correct fall-through instruction. Obviously, one could try to augment the trace with both the taken and not-taken sides of a branch or try to account for the wrong-path effects via approximations, but this introduces complexity.

2.2.2 Execution-driven simulation

Because speculative execution (and any other situation in which the target's execution timing can affect the events seen by the simulator) can complicate trace-based simulation, it is often easier to study these effects when a processor model is able to direct the delivery of instructions rather than passively receive them from a trace. Furthermore, a trace consisting of all microarchitectural events in a modern out-of-order superscalar system would require storing vast amounts of state. While it may suffice to capture only events of interest or apply an existing trace compression technique, execution-driven simulation can avoid the problem altogether and trade off speed to reduce storage requirements and increase accuracy.[1]

Rather than capturing significant events with a trace generator for later use, each instruction of a program is simulated as it is encountered, and a performance model provides timing information for that instruction. The next instruction to be simulated is determined by the simulation of the previous instruction. For example, a branch target address is determined by computation in the simulator (perhaps adding an offset to the current program counter); the address is not simply provided as input to the simulator. This allows the simulator to execute speculatively and out-of-order as long as it can commit in order and roll back from incorrect speculation. Execution-driven simulation differs from trace-driven simulation in that the former directly allows performance models to affect the arrival of the workload's instructions. With fast host computers and the distribution of an easy to use, parameterized simulator [14], execution-driven simulation surged in popularity beginning in the late 1990's and became the dominant approach in academic studies. However, the detail provided by execution-driven simulations comes at the cost of speed and development effort. Not only do we preclude the ability to reuse the work of fetching and decoding instructions, but we must also provide more accurate models of every structure in the processor. Usage of finite resources must be correctly controlled, and instructions stalled until their required resource becomes free. Data in caches and buffers can cause instructions to experience variable latency and should be modeled faithfully. This increased fidelity requires more host cycles and slows down simulation.

[1] What we now refer to as execution-driven simulation was originally termed *instruction-driven simulation* [28].

2.3 Sampling

This thesis is motivated by the fact that architects are rarely satisfied by the speed of their simulators. One way to increase speed is to perform less work by sampling, i.e., simulating only a portion of a benchmark. This section will discuss issues related to sampling. First, we describe two ways to advance a benchmark to sample points: 1) *online sampling* that alternates between two levels of detail until the experiment is complete and 2) a *checkpointing implementation* that warms-up state using a checkpoint generated from a previous run. The concept of checkpoints is discussed in detail. Next, we discuss the problem of cold-start effects present in both online and checkpoint-based simulation. By starting at a given point, one risks incorrect measurements unless structures in the computer are warmed-up (as suggested by Chapter 1's analogy of a marathon runner). Several solutions to the cold-start problem are reviewed. Finally, we discuss how samples are chosen.

2.3.1 Advancing to a sample

To advance to a sample *online* requires the construction of two simulators — or one simulator with two modes: a functional simulator used to quickly advance to sample points and a detailed simulator used to provide performance results for a sample point. Fast functional simulation advances the state of target until a sample point is reached. At the sample point, a detailed simulation begins. Detailed simulation is slower because it must model the system more accurately and collect useful statistics. The alternation repeats as often as necessary.

Even though functional simulation may be orders of magnitude faster than detailed simulation, online sampling methodologies usually require that the simulator spend the bulk of its time performing functional simulation. Alternatively, one may perform a preliminary functional simulation to generate *checkpoints*. Checkpoints store the state of the target prior to every sample point, amortizing the work of functional simulation. The detailed simulator is loaded from the checkpoint to measure the performance of target as it executes the sample. While checkpoint-based sampling can be fast once checkpoints have been created, one should not discount the utility of online techniques as they can be desirable for one-off studies or when the collection of checkpoints is difficult [85]. Nevertheless, we continue with an overview of checkpoints because, when applicable, they offer substantial speed benefits.

2.3.2 Checkpoints

One way to amortize the time needed to warm-up large structures is to draw from research in the area of application checkpointing. When one is running important programs that can take multiple days to complete, the ability to save to — and restart from — a checkpoint is essential. This protects the user from hardware failures, power outages, or unexplained crashes. Checkpoints are also useful for process migration, as in cases where a batch scheduler can more optimally allocate resources by moving a process from one machine to another.

Using a typical taxonomy, there are two methods for checkpoint and restart [13]. A System-Level Checkpoint (SLC) resembles a core-dump: a record of all programmer-visible state belonging to a process. With such a checkpoint, the program can be resumed at the moment the checkpoint was saved. SLC creation can be performed periodically and automatically by the batch queuing system, assuming the operating system provides support. The SLC is bound to the particular machine that created the checkpoint as it contains data structures specific to the architecture and operating system. Such checkpoints can be difficult to generate when a program has many open files, sockets, communicating processes, and/or threads. Projects such as Condor and CHPOX provide SLCs on existing operating systems with certain restrictions on which system-level facilities are supported [24, 109]. The SPRITE network operating system was designed from the ground up to support complete SLCs [83]. SPRITE used an SLC to migrate a process from a busy workstation to another networked workstation that was idle.

An Application-Level Checkpoint (ALC) requires that the programmer modify his application, saving the state of key structures and providing hooks that allow these structures to be filled from an ALC on disk. For highly-structured scientific studies, the ALC can be easy to produce: the checkpoint contains the state of the simulated system at a given point in simulated time. For example, it could contain the position and mass of all particles in an n-body problem, or it may store the current contents of a matrix in an FFT application. When global barriers are present, they can be used to force a checkpoint. To improve performance and limit the size of checkpoints, the programmer may provide hints which indicate potential checkpoint sites. Restoring multithreaded applications is especially difficult and requires care to avoid deadlock prior to checkpointing and at restart time [13]. The number of ALCs may be reduced by monitoring system-level performance. For instance, if the system is experiencing an unusually high level of disk or network traffic, the time to create the checkpoint may outweigh the time it would save if used for recovery. Alternatively, if an impending failure can be predicted at the system level, the value of the checkpoint is increased [80].

Checkpointing has a different purpose in the context of computer architecture simulators, where it can be used to reduce the runtime of a simulation rather than to provide fault tolerance. By capturing the state of a program at an interesting phase, we can avoid spending time simulating an uninteresting, unrepresentative, or well-understood phase. Alternatively, with many checkpoints saved throughout a program's execution, we can employ statistical sampling techniques or weighting of results to estimate a total.

The SimSnap project introduces a compiler which inserts checkpoint generation routines into application code [112]. SimSnap annotates code with hints to the compiler (pragmas) that either suggest or mandate a checkpoint. The compiler replaces the pragmas with checkpointing code. In addition, certain system calls, such as `malloc()`, must be replaced by checkpoint-aware versions so that the simulated application can easily restore its memory and continue execution. The checkpoints, which contain all programmer-visible state, can be

gathered on native hardware for use with a detailed simulator. Some perturbation of results is noted due to the injected code. In addition, the detailed simulator must still warm-up microarchitectural structures.

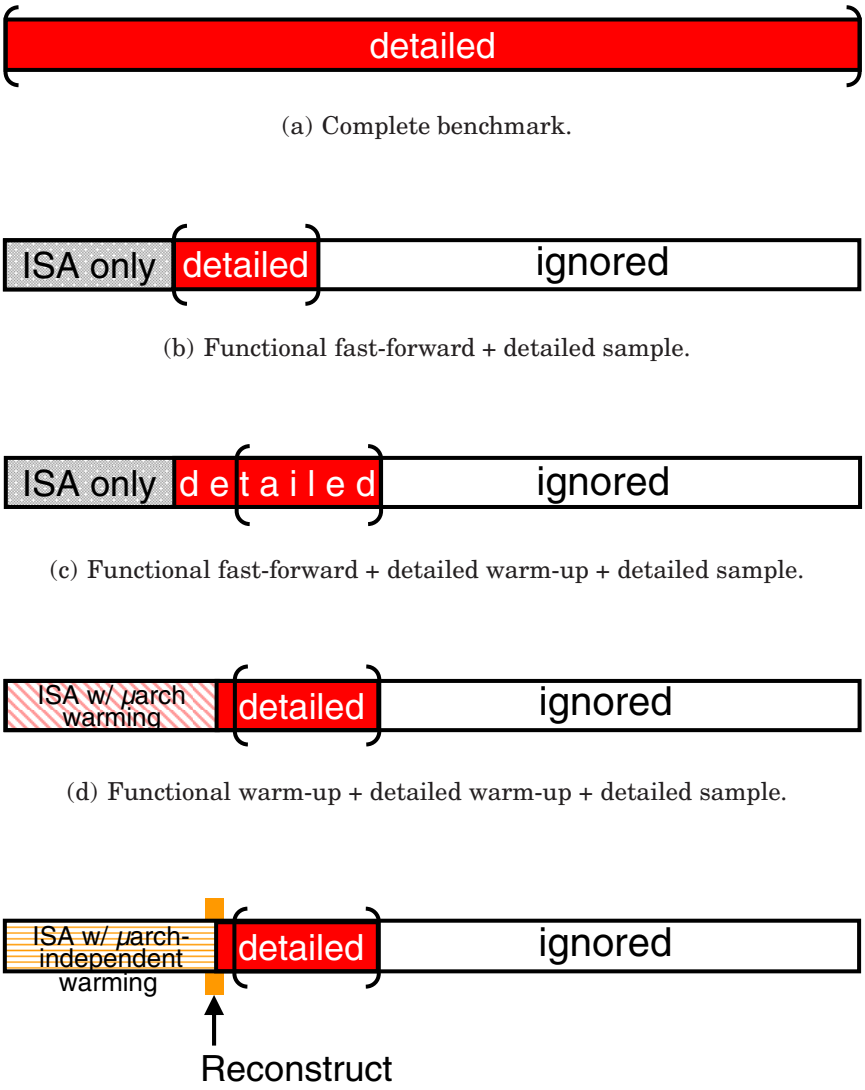
Binary-rewriting has been suggested as a way to create checkpoints without modifying benchmark source code [91]. In this system, the program binary is augmented with a prologue that contains the necessary stores to memory to create the memory checkpoint and load-immediates to fill the register file so that the program may be resumed at the checkpoint site. The system also reconstructs the effects of system calls that affect the system environment. For SPEC CPU2000 benchmarks, the amount of instruction overhead averaged less than 3% and required an average of 17 MB for data. The technique eliminates the need to fast-forward to a sample point, reducing runtime by a factor of 60 compared to ISA-only fast-forwarding. Such a system accounts for architectural state changes only, potentially requiring long warming of microarchitectural structures.

2.3.3 Avoiding cold-start effects

A crucial aspect to accurate sampling is to reduce cold-start effects. Figure 2-2 shows several options for avoiding cold-start effects beginning with 2-2(a), the infeasible option of running the entire benchmark in a detailed simulator. When we fast-forward to a sample point in an ISA-only mode, caches, branch predictors, and other microarchitectural structures are empty or invalid (Figure 2-2(b)). When detailed sampling begins, an empty cache will yield a large number of misses compared to one that is warmed-up with recently-accessed data. Likewise, all predictors will be untrained and unlikely to offer correct predictions.

Several techniques have been proposed to repair or warm-up cache state prior to measurement in order to obtain more accurate simulation results. *Trace stitching* uses the cache state from the end of a previous detailed sample period to approximate the initial cache state [2]. An alternative is to begin each sample with an empty cache and simulate memory references until a set is completely full, or *primed*. Once the set is primed, references to the set are allowed to contribute to cache statistics [58]. Or, one may merely use the first portion of a sample to warm the cache and collect statistics for accesses occurring during the second portion. For direct mapped caches, executing the first half of a sample with a detailed simulator and collecting statistics during the second half can be simpler and more accurate than the set priming technique [54]. Figure 2-2(c) shows the straightforward use of detailed warming applied after fast-forwarding to a sample point.

Memory Reference Reuse Latency (MRRL) can be used to bound the amount of detailed warming prior to a sample to achieve a desired accuracy [44]. By limiting warming to a "pre-cluster" region immediately preceding a sample point, it is likely that the data relevant to the upcoming sample will be brought into the cache, and older, irrelevant memory accesses can be ignored. MRRL refers to the number of instructions between a memory access to address A and the previous access to A. Applying MRRL to warm-up and sampling requires a pro-



(a) Complete benchmark.

(b) Functional fast-forward + detailed sample.

(c) Functional fast-forward + detailed warm-up + detailed sample.

(d) Functional warm-up + detailed warm-up + detailed sample.

(e) Microarchitecture-independent warm-up + reconstruction + detailed warm-up + detailed sample.

Figure 2-2. Techniques to avoid cold-start effects. Brackets indicate region where measurements are taken.

filing phase in which MRRLs are determined. With an instruction-based notion of latency, it is possible to run an application and gather a profile that applies to any cache configuration. A threshold, N, is chosen and a corresponding warm-up period, w_N, is calculated such that N% of the memory addresses accessed within a sample are previously referenced within the past w_N instructions. In a uniprocessor model without operating system effects, MRRL can remove 90% of the warm-up costs with less than 1% error in reported IPC. As structures grow larger and are duplicated in a multiprocessor, the remaining warm-up time may still be significant. Also, the reference patterns in a full-system multiprocessor simulation change non-deterministically with changes in the microarchitecture, which may make it more difficult to apply MRRL. For branch predictors with long histories, it is not obvious that an MRRL-based technique would suffice to capture relevant branches.

Detailed warming increases the time a simulator spends in its slowest, most-detailed mode. To reduce this time and address the cold-start problem, the SMARTS framework recently proposed *functional warming*, which simulates large structures (such as caches and branch predictors) during the fast-forwarding mode [127]. While performing functional warming, the function of the structures is simulated — not the structures' timing. As a consequence, the large components of the microarchitecture are already warmed-up at each detailed sample point, drastically reducing the amount of slow detailed warming prior to a sample. A small amount of detailed warming is used to fill smaller structures such as pipeline registers, buffers, and queues.

For a uniprocessor cache model, functional warming involves indexing into the cache, performing a tag lookup, implementing the cache replacement policy, and maintaining the state of each cache block (dirty, clean, or invalid). For branch predictors, history shift registers and tables of prediction counters are kept up-to-date on every branch. By introducing a slight overhead during fast-forwarding, functional warming shortens the detailed warming phase while providing accurate measurements. Figure 2-2(d) illustrates functional warming. SMARTS selects samples uniformly over an entire benchmark run. For uniprocessor models, as little as 0.1% of a benchmark must be run in a slow detailed simulator to produce accurate results. After verifying that samples from different stages of execution are independent (that is, they do not coincide with periodicity of the program), SMARTS applies well-known statistical results to estimate the true mean of a metric from the samples' mean. Sampling theory allows the estimate to be bounded by confidence intervals.

The MTR and BPC described in this thesis were created specifically to address the problem of avoiding cold-starts regardless of the microarchitecture under investigation. The MTR adds multiprocessor cache and directory support to prior work in uniprocessor warming, while BPC handles branch predictor tables. For comparison to other techniques, the use of MINSnaps is shown in Figure 2-2(e). A MINSnap is created during a period of warming that includes both ISA simulation and the updating of a microarchitecture-independent structure. This structure can be saved in a checkpoint or used immediately. A reconstruction

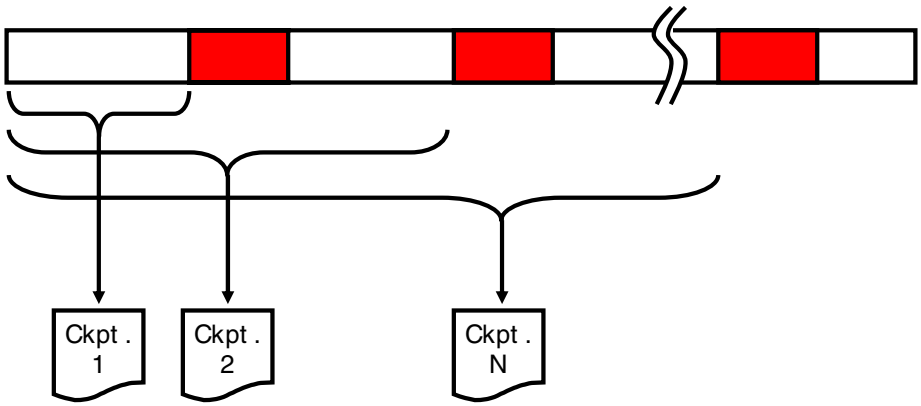


Figure 2-3. Checkpoint generation.

phase interprets the data in the MINSnap and fills in the concrete microarchitectural state of an experiment's target configuration prior to a detailed sample.

2.3.4 Contents of checkpoints

Checkpoints must contain all the information necessary to initialize a particular detailed sample. Figure 2-3 depicts the logical contents of each checkpoint, i.e., what portion of the program is represented by each checkpoint. Checkpoint 1 contains the state of the target after running a benchmark from its beginning until the first sample point. Checkpoint 2 contains the target's state as a result of running the program up to the second sample. Checkpoint creation continues through Checkpoint N. A naïve implementation would cause the size of the collection of checkpoints to grow quadratically, but it is possible to represent the collection in linear space. The nth checkpoint includes only the information that differs between itself and the checkpoint for sample $n-1$.

The specific contents of a checkpoint can include three types of state: architectural, microarchitectural, or microarchitecture-independent.

Architectural state. Checkpoints of architectural state can be created using simulation or by interrupting execution of the application on a real machine. Such a checkpoint contains only state defined by the target's ISA. This includes such data as the contents of logical registers (but not physical registers) and memory (but not cache). The checkpoints can then be used to initialize different machine configurations without repeating fast-forwarding. To reduce cold-start effects, microarchitectural state can be reconstructed using a detailed warming phase before results are gathered at each sample point, as shown in Figure 2-4. If the microarchitectural state is large, such as for caches and branch predictors, the time required for detailed warming can be prohibitive.

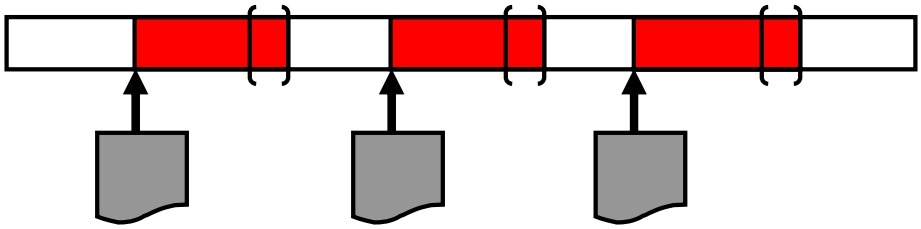


Figure 2-4. Checkpoints containing only architectural state require lengthy detailed warming.

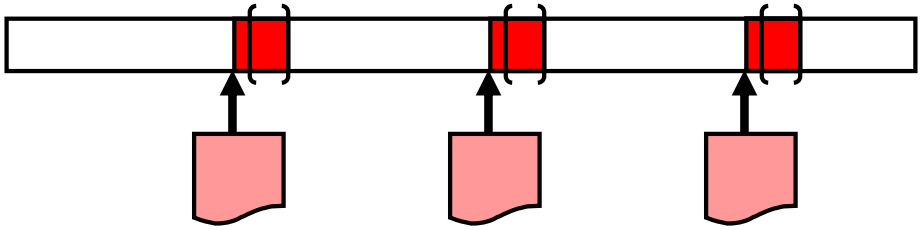


Figure 2-5. Checkpoints containing microarchitectural state require less detailed warming but limit versatility.

Microarchitectural state. Microarchitecture refers to the elements of a computer's design that are not exposed to the programmer. In other words, a single architecture can be realized with many different microarchitectural implementations.

If the checkpoint stores information about microarchitectural structures (e.g., contents of caches, predictors, and queues), it avoids a time-consuming warm-up phase as shown in Figure 2-5, but the checkpoint becomes less versatile: it can only reconstruct structures similar to those that have been saved. This can be undesirable as it is often the microarchitecture that is varied across experiments and checkpoint regeneration is required every time a microarchitectural feature is modified. Checkpoint regeneration can be costly: some industry development groups report detailed warming runs require weeks when starting from stored architectural checkpoints [37].

If one knows in advance the parameters of the microarchitectural structures of interest, multiple simulations can be used prior to generating a checkpoint for every interesting structure. This library of checkpoints can be used repeatedly to drive various detailed simulations. However, storing checkpoints for many structures at many sample points can require large amounts of storage.

Microarchitecture-independent state. Modern directory-based, cache-coherent multiprocessors have an ever growing quantity of microarchitectural state, including the directory and multiple large caches. The long histories of these structures make detailed warming from an architectural checkpoint impractical, and their size can make multiple microarchitectural checkpoints infeasible due to storage costs. Ideally, the checkpoints would be

microarchitecture-independent, yet allow fast reconstruction of any desired microarchitectural configuration.

A checkpoint, or snapshot, containing microarchitecture-independent state can be a useful compromise between the techniques described above. While it cannot match the speed of directly restoring a microarchitecture-dependent snapshot, a microarchitecture-independent snapshot (MINSnap) provides the versatility of a snapshot that contains only architectural state. Instead of lengthy detailed warming, a MINSnap uses a reconstruction period to fill in the contents of microarchitectural structures (Figure 2-6). MINSnaps support microarchitectural exploration with a single set of stored checkpoints.

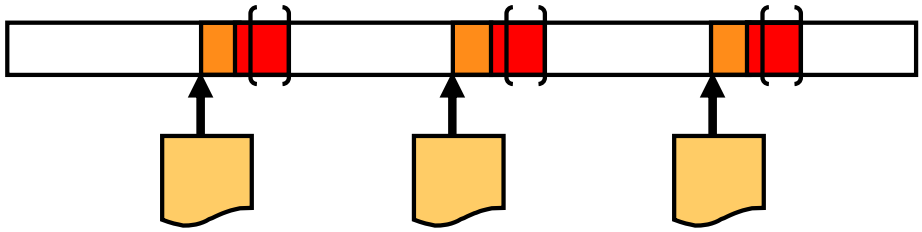


Figure 2-6. Microarchitecture-independent snapshots (MINSnaps) allow long periods of detailed warming to be replaced by a short reconstruction period.

2.3.5 Choosing sample points

Apart from addressing the issue of cold starts, the architect has several options for choosing which segments of the program comprise the sample. Figure 2-7 shows several of these options; the color of the figure differs from that of Figure 2-2 to reinforce the distinction between the cold-start problem and the sample location problem. We rarely have time to run an entire program as in Figure 2-7(a). A simple shortcut is to take a single short sample at the beginning of the program (Figure 2-7(b)). Unless the rest of the program behaves in a similar fashion, this single sample approach is likely to give inaccurate results. For programs that begin with a distinct initialization phase, we can try executing a portion of the program that occurs after the initialization phase, as shown in Figure 2-7(c). This only provides accurate results when the program has just two distinct phases. More complex programs require more sophisticated sample selection.

Runtime of some multithreaded programs may be estimated by taking detailed samples from only those processors comprising the critical path [43]. Other portions of the program do not contribute to the runtime and need not be sampled. When parallel programs have no clear critical path or when more fine-grained, overall statistics are required, other techniques may be required.

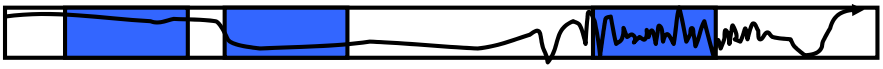
The rest of this section presents prior work in two areas: simple random sampling and phase-based sampling.

(a) Complete benchmark.

(b) Single sample.

(c) Skip + single sample.

(d) Simple random sampling.



(e) Phase-based sampling.

Figure 2-7. Choosing sample point(s).

Simple random sampling. Statistical sampling of traces can be used in the same fashion as drug trials or election exit polls [26, 58]. A random subset of the population is polled; the larger the subset, the more accurate the prediction. In practice, the subset may be quite small and still yield accurate results. Statisticians can bound their prediction within a confidence interval that denotes that the true outcome lies between two values with a specified measure of confidence. In other words, results can reliably be reported as "$\pm x$ with y% confidence." Confidence intervals tighten as the number of samples is increased and/or the variance between samples shrinks. This methodology can identify cases in which more samples need to be gathered in order to reach a certain confidence in the reported range. Alternatively, more samples can tighten the estimated range for a given confidence.

Figure 2-7(d) shows samples taken randomly over the course of a benchmark's execution. Typically, less than 1% of instructions need to be simulated in detail to obtain accurate results. With online sampling using SimpleScalar tools, this leads to speedups of 35–60× for the SPECCPU benchmarks, depending on the target configuration [127]. Simulating each sample in parallel can further reduce the amount of time required to achieve a result [61, 125]. When it is not feasible to gather random independent samples, the results of sampling must be validated with non-sampled experiments to ensure no bias is present in the sample selection.

Usually, tightening a confidence interval requires additional samples. However, for some applications, the presence of multithreading is sufficient to reduce the number of required samples [35]. When multiple processors are working on different portions of the same code, the effect of any single processor is muffled by its companions. This reduces variance of the samples resulting in fewer samples required for a given confidence interval. When the processors are coordinated, as is likely for barrier-based scientific applications, a performance metric for any single processor looks similar to its peers, and the variance is not lowered.

Phase-based sampling. An alternative to simple random sampling is to choose samples that are known to best represent program behavior. For example, a scientific benchmark may begin with an initialization phase during which a dataset is read. It may continue with processors performing a computational kernel in parallel followed by a reduction or sharing of information. This work-reduce pair of phases may be repeated many times before a concluding phase that prints the results. To measure the entire program, it is important to observe each of its phases, but it may not be necessary to repeatedly measure a phase that has similar characteristics during each of the 1000 times it occurs.

SimPoint is the name of a popular approach to phase detection and sampling [98]. With SimPoint, a program is first run in a fast, functional simulator to gather basic block execution histograms. One histogram, or basic block vector (BBV), is gathered for each fixed-length interval of instructions. SimPoint seeks out portions of the program whose basic block distribution mirrors the BBVs found in the program as a whole. Rather than compare every interval's BBV to every other interval's, SimPoint uses several optimizations including

a projection to reduce the dimension of the BBV and a clustering algorithm to find similar intervals. SimPoint can produce a single sample that is most representative of the entire benchmark, but it is more effective when several phases are identified and samples from each phase are weighted according to each phase's frequency. Figure 2-7(e) shows a benchmark with three distinct phases. Each phase is sampled. The second phase is longer, so more weight will be given to its sample.

Phase-based sampling is faster than simple random sampling because the amount of detailed simulation is limited to a fixed number of samples, each of which is taken from a representative phase. However, it lacks formal confidence intervals and phases must be rediscovered when programs are recompiled. There is a possibility that different occurrences of the same BBV will have different performance characteristics due to microarchitecture, but one cannot determine this difference with BBV-based phase detection. Though phases usually persist across microarchitectural changes [105, 85], it could be difficult to use BBVs to detect phases accurately in programs with code signatures that are susceptible to change in the face of microarchitectural changes. Spinlocks in multithreaded code are a construct that can change a BBV depending on system timing.

Another way to detect phases is to look for similarity in a metric such as IPC [105]. Similarity of distributions was proposed to identify phases more accurately than a comparison of mean values. While the absolute value of the metric may change due to microarchitecture changes, phase boundaries tend to persist despite changes to the microarchitecture.

When a program is run on an SMT processor, a mix of SimPoint and instruction throughput during a detailed sample interval can be used to guide fast-forwarding to the next interval or even to perform a purely analytical simulation using a Co-Phase Matrix [121]. The Co-Phase Matrix contains per-thread performance information for every combination of overlapping phases. For example, it could contain an entry that indicates "while thread 0 is in region a and thread 1 is in region x, thread 0 completes 2 instructions per cycle and thread 1 completes just 1 instruction per cycle." Phases for each thread are identified with SimPoint and the combinations simulated together to generate the Co-Phase Matrix. Once an entry for a co-phase exists, it can be used to guide fast forwarding. The number of instruction in a thread's phase combined with the IPC from the matrix can tell us the number of cycles that will elapse until the next co-phase. No actual simulation takes place once the matrix is populated. For workloads with a large number of threads and/or many phases, the number of combinations may be prohibitively large. A dynamic approach to filling the Co-Phase Matrix is proposed in which starting points are chosen and the matrix is populated with performance observed during simulation. Checkpoints are recorded at the beginning of each thread's SimPoint. A checkpoint from each thread may be combined to produce per-thread performance metrics subject to the resource constraints of an SMT. The proposed checkpoint format includes *Memory Hierarchy State* that is independent of cache configuration, but does not include a MINSnap for branch predictors [119, 120].

2.4 Simulation acceleration

There are several techniques for speeding up simulation that are orthogonal to the sampling approaches discussed above. This section reviews some of these techniques, which may be used in concert with sampling.

2.4.1 Accelerated instruction emulation

As computers became more complex, and simulation became relatively slower, researchers proposed *direct execution* to increase simulation speed [28, 126]. In direct execution, instructions from the simulated program are executed using the host system's native instructions rather than emulating the operation of each instruction. It is only necessary to enter the simulator to update performance statistics on branches or at communication points.

The Wisconsin Wind Tunnel project appropriated error correction bits in the host system's hardware, and generated an "error" whenever the host accessed memory (on behalf of the target) that was not in the target system's cache [90]. The simulator provided a custom error handler on the host that could react to and model target memory references. A second generation of the Wind Tunnel project took a more portable, but slower, approach: statically modifying code, replacing loads and stores with calls into the simulator to handle simulation of memory events [76]. Since switching between the target and host is kept to a minimum for performance reasons, it can be difficult to accurately model timing of advanced targets. When one is not concerned with the precise timing of instructions, direct execution can lead to speedups of up to 3.6× over similar execution-driven simulators with less than 3.9% error [34]. In a multiprocessor simulation, direct execution must be interrupted whenever the simulated processes interact [29]. The Time Warp mechanism, while not a direct execution simulator, strives to speculatively continue through synchronization points. It rolls back to correct errors if synchronization violations are detected [49].

More recently, the PTLsim project has advocated using direct execution for fast-forwarding prior to detailed sample points [128]. A target application runs natively until a breakpoint triggers the detailed model. Even during detailed modeling, the simulator can reap speed benefits because its target and host use the same ISA. Target instructions can be emulated with hand-coded assembly fragments and target system calls can be executed directly on the host OS. The newest version of the simulator, PTLsim/X, has leveraged the popular Xen *hypervisor* [129]. This allows the simulator to run at the highest privilege level, providing a virtual processor to the target OS. At this level, both the target's operating system and user-level instructions are modeled by the simulator, and it can communicate with Xen to provide I/O when needed by the target OS.

An alternative to direct execution is used by Simics [68], a full system simulator which runs unmodified binaries with operating system support. Originally, Simics used an automatic, profile-driven code generator to produce optimized C code for emulating target instruc-

tions [59]. The emulation was statically defined but by separating decoding and execution, specialized service routines could be installed to optimize speed after the initial decoding of an instruction. For example, if a general-purpose `add` instruction was decoded and found to have matching source and destination registers and "1" as an immediate operand, its service routine need only increment the register. There is no need to include an instruction to extract the constant "1." Furthermore, the simpler increment code may allow the compiler to perform constant propagation or common subexpression elimination. New versions of Simics [38] as well as QEMU [10], use binary rewriting. Binary rewriting is the process of dynamically translating sequences of instructions into functionally equivalent, but optimized or managed, sequences of instructions. Examples of optimizations include: removing the computation of condition codes when they are unneeded; simplifying address generation when the segment base is zero; and chaining basic blocks together [10].

2.4.2 Statistical/synthetic simulation

Another approach to reduce runtime is by creating a synthetic instruction stream that is statistically similar to an actual program [79, 82]. This methodology requires an initial functional simulation to capture the program's inherent characteristics, such as opcode mix, dependencies, locality, and predictability. Performance can be extrapolated through statistical simulation by generating small, statistically similar traces to drive a slower detailed simulator. In this context, statistics are not used for sampling (as in Section 2.3.5), but to ensure that the synthetic program is statistically similar to the original. With respect to multiprocessors, a 10–15% error in instruction throughput was observed depending on workload [79], though trends observed with the synthetic workload follow those seen in a detailed baseline model.

2.4.3 Parallel hosting

A common way to explore a design space is to select a few choices for each of several design parameters and run an experiment for each combination in the cross-product of these parameter settings. When multiple simulation hosts are available, each host can be delegated a small number of experiments and the independent computation can occur in parallel. This technique increases simulation bandwidth but does not address latency. Unfortunately, the growing complexity of computer architectures means that the latency, or end-to-end runtime of each experiment, is just as important. Every day spent waiting for a batch of simulations to complete is one less day available for running a new suite based on feedback from the previous simulation. The need for feedback and inspiration from a current experiment makes it difficult to maintain a full pipeline of experiments and motivates techniques to reduce simulation latency.

One may also exploit parallel host processors to speed up the simulation through parallelization of an individual experiment. For example, the Wisconsin Wind Tunnel takes

advantage of a parallel host to simulate a parallel machine, using portable message passing and synchronization directives rather than machine-specific ones [76]. Mermaid distributes target processor simulation across several machines in a cluster and forces communication to proceed through a sequential model [88]. A Blue Gene simulator utilizing a parallel execution model was devised to simulate the IBM Blue Gene/L supercomputer, and others of its genre, using existing parallel hardware [132]. This simulator relies on language and runtime support to limit the overhead of correcting causality violations. BGLsim, another Blue Gene simulator, overlays Blue Gene models on a Linux cluster, relying on message passing to handle coherence [18].

Experience with the Wisconsin Wind Tunnel II was reflected in the design of Intel's ASIM simulator, allowing it to be parallelized over a summer several years after it was introduced [8]. ASIM is a modular performance model framework in which parts of a computer system are represented by reusable modules connected by ports [36]. By insisting that inter-module communication occur only through modified ports and allowing modules to be invoked in distinct threads, the new parallel framework allows each simulated CPU to execute on a distinct host processor.

2.4.4 Hardware simulators

Software simulators are popular because they offer flexibility and the ability to observe every aspect of a computer's operation. Many have the additional benefit of running on inexpensive desktop computers and servers. These simulators tend to run at 10s of thousands to 10s of millions of instructions per second depending on level of detail and speed of host hardware. Hardware runs much faster (billions of operations per second), so it can be useful to use reconfigurable hardware to prototype new designs. When a design is too complicated to fit on a single field-programmable gate array (FPGA), it may be prototyped by ganging together many FPGAs [41, 42, 123]. As high-gate-count FPGAs with fast I/O become more affordable, researchers have ramped up work in hardware-assisted simulators to allow simulation speed of 10s-100s of millions of instructions per second [6, 20, 21, 47, 81]. Many flexibility and observability issues must be addressed before such systems become popular. While it is hard to compete with the low cost of software simulation, it is not hard to imagine useful hardware simulation platforms whose expense is justified by the speed benefit over current software-based tools.

These are exciting prospects but are still in early stages of development, so I believe software simulation will still remain popular. While the techniques presented in this thesis should be applicable to hardware-assisted simulators, the thesis focuses on current and future software frameworks.

2.5 Summary

Though it is impossible to include and explain every publication related to a topic as broad as computer architecture simulation, at this point the reader should possess sufficient background to place the remainder of the thesis in context. We have reviewed both trace-driven and execution-driven simulation and seen efforts to increase their speed. The most-recently published research in this area seeks to apply and refine older sampling techniques to accelerate a current generation of execution-driven simulators.

We have also presented several techniques for accelerating simulation that are complimentary to sampling. *Accelerated instruction emulation* can be especially helpful for advancing to samples and/or creating snapshots for initializing samples. However, accelerated instruction emulation is only one aspect of minimizing overall simulation time. In a multiprocessor simulation, the memory operations, not the instruction emulation, constitute the largest obstacle to high speed simulation of a parallel processor [68]. Therefore, techniques such as direct execution and binary rewriting are not a panacea.

We have reviewed prior work in the area of *parallel hosted* simulation. In Section 3.2, we will outline how the MTR is well-suited for such parallel simulation. Branch prediction does not have the synchronization needs of shared memory, so the BPC may be used on a parallel host as well.

Statistical techniques are related to our work in that they seek to provide quick answers to computer architecture questions, but they represent a significantly different strategy — usually employed prior to the full-system studies that are the setting for the MTR and BPC.

Our work makes sampling simulations more versatile, easing the investigation of multiple microarchitectural targets from a single summary of execution. In Chapter 3, we introduce a method for storing the state of a cache-coherent multiprocessor memory system in a microarchitecture-independent fashion. Chapter 4 uses a lossless trace compression technique to represent branch predictor state. With these MINSnaps, one can save space and amortize the lengthy work of snapshot generation across many microarchitectural experiments, shortening the time required to design and evaluate computers of the future.

Chapter 3

Memory Timestamp Record

In this chapter, we present a fast and accurate technique for initializing the directory and cache state of a multiprocessor system based on a novel software structure called the *memory timestamp record* (MTR).[1] The MTR is a versatile, compressed snapshot of memory reference patterns which can be rapidly updated during fast-forwarding, or stored as part of a checkpoint. When advancing to sample points online, the MTR simply records the time of every processor's last access to every memory block instead of maintaining the directory and cache state (as with functional warming). This bookkeeping adds little overhead to functional simulation, yet the MTR can quickly and accurately reconstruct cache and directory state largely independent of size, organization, or protocol. Once the memory system has been reconstructed, detailed performance simulation observes the microarchitecture's effect on timing. We show that, with our implementations, the MTR achieves an average speedup of 1.19–1.45 over conventional fast functional warming (FFW) for a single cache configuration. Additional speedup is possible when multiple different cache organizations are reconstructed at each sample point. We show we can simulate several different MTR-initialized configurations in the same time as one run with FFW. In addition, both MTR updates and MTR cache reconstruction are highly parallelizable, thus easily supporting parallel-hosted simulation. When used as a checkpoint, the MTR offers the same microarchitecture-independence as prior work based on stack algorithms, but it requires less space and has no upper bounds on target cache sizes.

3.1 MTR Design

Memory reference traces are a microarchitecture-independent summary of one possible thread interleaving of a multithreaded program. Replaying a trace can serve as a straightforward way to update the directory and caches, avoiding cold-start effects during sampling. With in-

[1] The material in this chapter is based on the joint work of Kenneth Barr, Heidi Pan, Michael Zhang, and Krste Asanović. This work originally appeared in the International Symposium on Performance Analysis of Systems and Software held in March of 2005 [9].

finite disk space and simulation time, all the memory accesses observed prior to every sample point could be stored. This trace of accesses could then be replayed to warm-up any target memory system prior to a detailed sample, but this has extensive storage requirements and would be quite slow. The MTR instead exploits properties of time-based replacement policies and cache coherence to form a small, self-compressing structure which sacrifices some accuracy for low storage requirements and high speed. This section describes the conceptual structure of the MTR and how it accomplishes the goals of high speed, low space, and significant versatility.

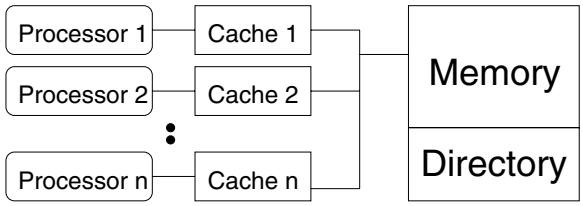


Figure 3-1. A simple symmetric multiprocessor (SMP) target system.

To illustrate the operation of MTR, we use the simple symmetric multiprocessor (SMP) model shown in Figure 3-1 as our simulation target. Every processor has a local cache, each having the same cache parameters (e.g. size, associativity) and using an LRU replacement policy. The memory uses a centralized full-map bit-vector directory and the MSI write-back invalidation protocol to support sequential consistency. The directory is always notified when dirty blocks are written back, whereas clean blocks may be silently evicted without informing the directory. The same block size is used by both caches and memory. Alternative cache organizations, coherence policies, and replacement policies are explored in Section 3.2.

3.1.1 MTR structure

Rather than functionally simulating a cache to keep it warm, the MTR performs a simple and fast table update on every memory access. This reduces the amount of work per memory access; we defer cache state reconstruction until it is time to prepare for a detailed sample. The key observation is that directory and cache state can be reconstructed if, for each memory block, we know about each processor's latest accesses to that block and the relative order of these accesses across processors. During MTR creation, each simulated processor reads and writes a shared "magic memory" for instant resolution of loads and stores. The MTR also captures the most recent memory accesses using the structure shown in Figure 3-2. The MTR has an entry for every memory block. Each block's entry contains an array of read timestamps, one per processor, indicating the last time each processor read the block. Additional fields record the identity of the last processor to modify the block and the timestamp of the write. Note that the array of read timestamps constrains the microarchitecture-independence of the MTR; it can only represent microarchitectures with a predetermined number of CPUs.

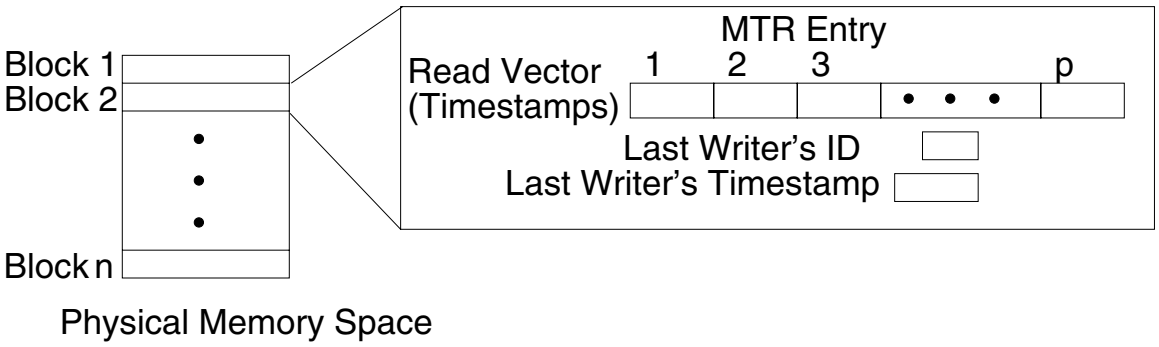


Figure 3-2. A Memory Timestamp Record.

3.1.2 MTR creation

Whenever any processor issues a read or write request prior to a sample point, the `Update` algorithm shown in Figure 3-3 is used to update the MTR.[2] A read request for a block of memory will place a timestamp in the read timestamp field corresponding to the block's address and reader. A subsequent read of the same block by the same processor will overwrite the previous read's timestamp. A write request updates the MTR entry's writer ID and timestamp. When advancing to samples online, we can also execute `Update` during detailed simulation to keep the MTR consistent with directory and cache state at all times. The lightweight MTR update has little effect on detailed simulation speed.

```
Update(address, isStore, cpu) {
   time++
   MTR[address].readers[cpu] = time
   if(isStore) {
      MTR[address].writer = cpu
      MTR[address].writetime = time
   }
}
```

Figure 3-3. MTR updates during fast-forwarding.

Note that we update both the read and write timestamp of a block when a write occurs. Figure 3-4 illustrates the reason. Assume the caches in the figure are four-way associative with LRU replacement, and assume that all of the depicted memory accesses map to the same set. Processor 1's write to block *b5* causes *b1* to be evicted, but the write request is overwritten in the MTR by Processor 2's write to the same block. Information is lost that would be helpful for establishing *b1*'s eviction. This information is retained by updating the read vector along with the write timestamp. In this specific example, the read vector update allows the MTR to have knowledge of P1's access to *b5* for reconstructing the eviction. Marking writes as read-modify-writes preserves the correctness of the MTR reconstruction algorithms because the

[2]To show the MTR algorithms, we use a C++-like pseudocode, using dot notation to indicate both attributes and method calls on objects.

final directory and cache state for read-modify-writes is indistinguishable from the final state for writes. If no other processor accesses the same block afterwards, the write timestamp indicates that the block is modified; the simultaneous read is ignored. If another processor writes to the same block, the previous owner's copy becomes invalid, regardless of whether it was last read or last written. Similarly, if another processor reads the block, the previous owner's copy becomes clean-shared under the MSI protocol.

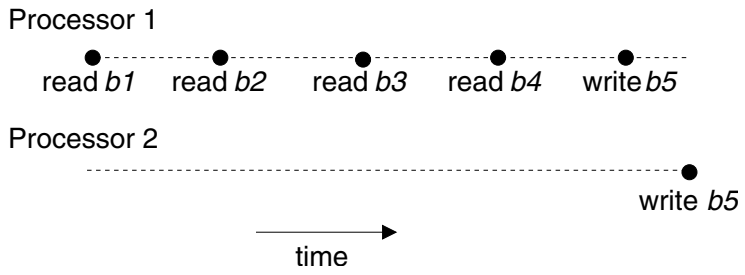


Figure 3-4. Scenario illustrating why write accesses are recorded in the MTR as read-modify-write. Assume a four-way associative cache with LRU replacement. Blocks *b1*-*b5* map to the same set. We need to retain the time of P1's write to *b5* to reconstruct the eviction of *b1*.

3.1.3 Cache reconstruction

At each detailed sample point, the cache and directory state must be quickly reconstructed. After choosing the size and associativity of the target cache(s), cache reconstruction is split into two phases. First, we filter the latest memory accesses recorded in the MTR to determine the subset that may be in the cache based on cache size and associativity. Second, we examine inter-cache relationships to determine the validity of each cache block and whether it is dirty.

To determine the cached subset, we observe that for k-way set-associative caches, an LRU policy dictates that only the last k accessed blocks remain cached in each set. To compare memory accesses that map to the same cache set, we reorganize the information in the MTR into a separate structure called the *cache set record* (CSR), shown in Figure 3-5. The CSR contains an entry for each set in every cache, holding a timestamp-sorted list of the k most recent memory accesses to that set. Figure 3-6 describes how to fill the CSR by sorting all the memory accesses mapping to the same cache set. The `Insert` function (not shown) kicks out the oldest entry if the array would overflow, leaving the k most recent accesses. To avoid inserting both a read and write timestamp for the same block into the CSR entry, the routine checks if a processor was the last writer. Note that in `CoalesceCacheblocks`, we insert all memory accesses into the CSR, whether they are valid or not. Although some of these cached blocks may be invalidated later by the cache protocol, they were valid when brought into the cache and potentially caused evictions. The MTR only contains entries for memory blocks that have been accessed, so the runtime of this procedure is $O(\text{touched lines} \times \texttt{NUMCPUS} \times k)$.

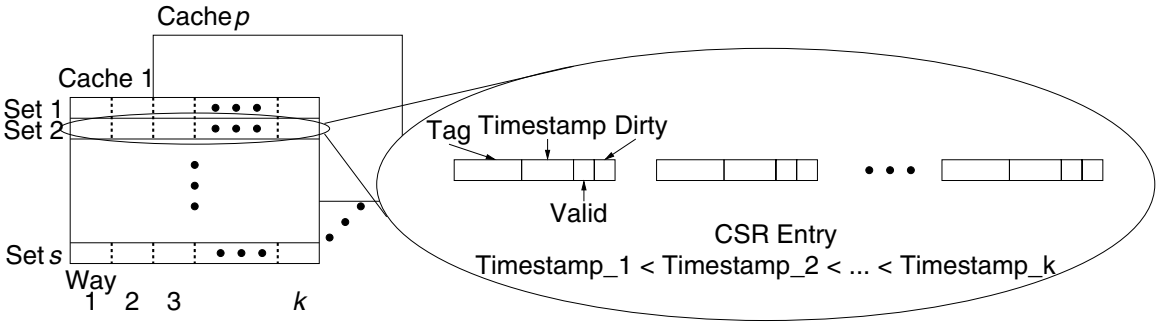


Figure 3-5. A cache set record (CSR).

```
CoalesceCacheblocks(CSR, MTR) {
   for each entry, i, in MTR {
      set = i.address >> SETSHIFT
      for(p = 1 to NUMCPUS) {
         if(i.readtime[p] is valid and p != i.writer) {
           Insert(CSR[set][p], i.tag, i.readtime[p])
         }
      }
      if(i.writetime is valid) {
         Insert(CSR[set][i.writer], i.tag,
                MAX(i.readtime[i.writer], i.writetime))
      }
   }
}
```

Figure 3-6. Building the CSR from the MTR: First pass to dump data from the MTR to the CPUs' cache set record (CSR). In each CSR entry, we keep an array of size ≤ *k* (cache associativity) that is sorted by timestamps. The `Insert` function kicks out the oldest entry if the array would overflow, leaving the *k* most recent accesses.

```
FixupCaches(CSR, MTR) {
   for each cached block b in CSR {
      lastwriter = MTR[b.address].writer
      lastwritetime = MTR[b.address].writetime
      if(b.timestamp < lastwritetime) {
         b.valid = false      /* invalidated by later write      */
         b.dirty = false
      }
      else {
         b.valid = true
         if(IsCleanShared(MTR[b.address]) {
            b.dirty = false   /* downgraded write or cached read */
         }
         else {
            b.dirty = true    /* cached write                    */
         }
      }
   }
}
```

Figure 3-7. `FixupCaches` reconstructs cache valid and dirty bits by examining inter-cache relationships. A block is valid if it was accessed during or after the last write.

In the second phase, we step through the CSR to determine the valid and dirty bits of these cached blocks using the algorithm `FixupCaches` shown in Figure 3-7. The state of a cached block is dependent on other processors' accesses to the same block, so we need to refer back to the corresponding MTR entry to determine when it was last read and written. A cached block can only be valid if it was cached upon or after the last write to the block; cache blocks present before the last write would have been invalidated to preserve consistency. A modified cached block only remains dirty if no other processors have read the data since the modification, otherwise it would have been downgraded to shared status by the subsequent read request. To check for such downgrades, we use the simple `IsCleanShared` test shown in Figure 3-8. Due to memoization, `IsCleanShared` runs in constant time for all but the first call per block (for which it is O(`NUMCPUS`)). `FixupCaches` invokes the linear-time `IsCleanShared` one time at the most. The remaining calls — as many as `NUMCPUS` − 1 — are serviced in constant time. This results in a worst case runtime for `FixupCaches` of O(blocks-per-cache × `NUMCPUS`).

3.1.4 Directory reconstruction

MTR directory reconstruction is similar to cache reconstruction, as shown in Figure 3-9. If a block is read but never written, or read by another processor after the last write, the block is shared. The sharers consist of the last writer (if any) and all subsequent readers. Although some of these sharers may have already evicted their clean copy of the block, they remain in the sharing vector under the silent drop policy, whereas the directory is always notified of

```
IsCleanShared(MTRentry i) {
   if(!i.clean_shared_memo) {
      i.clean_shared_memo = true;
      if(i.writetime is valid) {
         for(p = 1 to NUMCPUS) {
            if(i.writer != p and readtime[p] > i.writetime) {
               i.clean_shared = true; /* read after write */
               return i.clean_shared
            }
         }
         i.clean_shared = false;      /* not shared, still dirty */
      }
      else {
         i.clean_shared = true;       /* never written */
      }
   }
   return i.clean_shared
}
```

Figure 3-8. The `IsCleanShared` procedure returns true for blocks at which a read has occurred since the last write. It also returns true for blocks that have not been written. Otherwise, it returns false.

```
CreateDirectoryFromMTR(directory, MTR) {
   for each entry, i, in MTR {
      if(IsCleanShared(i)) {
         directory[i.address].state = Shared
         for(p = 1 to NUMCPUS) {
            if(i.readtime[p] >= i.writetime) {
               directory[i.address].addSharer(p)
            }
         }
      }
      else if(i.writetime is valid) {
         if(IsValidInCSR(i.address, i.writer)) {
            directory[i.address].state = Modified
            directory[i.address].owner = i.writer
            /* but if writer reads again, then this is an ambiguity */
         }
         else {
            directory[i.address].state = Invalid
            directory[i.address].clearSharers()
         }
      }
   }
}
```

Figure 3-9. Reconstruct directory state in a system with silent evictions. Note, `IsValidInCSR` returns true if the modified copy is still in the last writer's cache, and false if it has been evicted and written back.

dirty writebacks. Thus, we must verify that a dirty copy is still in the cache before marking it modified, otherwise it is invalid. All unrequested blocks are uncached and marked invalid in the directory.

When performing online sampling, the MTR contains only those memory accesses that have occurred since the last sample point. Thus, when reconstructing the caches and directory, a block that is not in the MTR is not necessarily uncached. Reconstruction during online sampling requires merging existing concrete structures with data from the MTR. Consider a block that becomes shared prior to MTR creation and not accessed during MTR creation. We must retain the state of the block rather than assuming its absence from the MTR implies that it is invalid. When sampling with checkpoints, each checkpoint is comprised of all prior MTRs. By successive reconstruction, no information is missing and the issue of merging is moot.

The runtime of `CreateDirectoryFromMTR` is $O(\text{touched lines} \times \texttt{NUMCPUS})$ due to the loop used to create a sharing vector and the invocation of `IsCleanShared` for each block. While the MTR algorithms are sensitive to the number of CPUs, until we begin simulating very large multiprocessors, we are most sensitive to the number of uniquely touched lines prior to each sample. In contrast, a functional warming simulation performs an $O(1)$ lookup on each hit and a $O(N)$ invalidate in the worst case when it must invalidate all sharers. The observed speed for our problem size depends on the constants hidden by Big-O notation; as we will see in Figure 3-17, the amount of work-per-access done by functional warming outweighs that done by the MTR. When the number of references represented by the MTR exceeds the number of uniquely touched lines, as it should in the presence of locality, the MTR should outperform FFW.

The reconstruction time would be extremely high if we had to examine each block of memory to reconstruct the entire directory and cache state for every detailed sample. Luckily, only a small subset of the memory locations and cache entries are accessed in each fast-forwarding period, so we only have to apply the reconstruction algorithms to this subset. We add two levels of valid bits to the MTR, which are updated during its creation, to track which memory regions and which blocks within these regions have been touched. The MTR is divided into "pages" of records. The page size, which is independent of the virtual memory system's page size, is chosen to keep the first-level table small and cacheable. For instance, if a four-CPU target uses 32 B blocks and a main memory of 64 MB, a 4096-entry page would result in 512 flags in the first-level table. If an entry in the first-level table is marked valid, then at least one of the memory blocks on its page has been accessed; if the page is marked invalid, then none of its members have been modified. With this organization, we can quickly skip large contiguous areas of untouched MTR entries during reconstruction.

The CSR is cleared before reconstruction. After reconstruction, it contains only those memory accesses that have occurred between samples, and it reflects the portion of the cache state that has been changed since the last sample. The CSR must be merged with the cache

state from the end of the last detailed sample (which has become stale by the time the next sample is reached). During the merge, we must also update the directory to reflect involuntary eviction of stale cache entries. This merging process is described in Section 3.4.

The MTR is well suited to parallel-hosted simulation in which each simulated CPU runs in its own thread. A slightly modified MTR in which each processor has its own write timestamp field would permit fast simulation without synchronization for every memory operation. While the speed of the host processors will dictate the relative order of memory requests, the only concurrency constraint is that application-level atomic instructions must be implemented with atomic accesses to the simulated shared memory to ensure legal execution orderings. To create the MTR, each processor would make fast updates as before, but the last writer would be determined at reconstruction time by comparing all CPUs' write timestamps. During parallel reconstruction, the shared MTR is read-only and may be accessed simultaneously by many threads. The writeable CSR is divided into per-CPU sections that can be written without the need for locks. A barrier is needed between the coalesce and fixup stages as `FixupCaches` assumes the contents of each cache have been completely determined.

3.1.5 Handling ambiguous cases

There are scenarios where the MTR timestamp information is insufficient to distinguish between multiple different directory and cache states. For example, consider Figure 3-10(a), where a processor writes then reads a block without other processors' interference. Given the write and read timestamps, one can infer either of the following scenarios: (1) the read request results in a cache hit and the data remains modified, or (2) the data is evicted and written back in between, so the read request brings the data back in a clean shared state. Another ambiguous example is depicted in Figure 3-10(b). The MTR tells us that the block is first written by processor *p1* then read by processor *p2*. After filling the CSR, we can determine that the block has been evicted from *p1*'s cache, but the CSR does not tell us when the eviction took place. If *p1* evicts the block before *p2* reads it, *p1*'s copy would have been dirty, so *p1* would have to write back the data. On the other hand, if *p1* evicts after *p2*'s read, *p1*'s block would have already been downgraded to a clean shared status, so it can be silently dropped. The directory would include *p1* in the sharing vector in the second case, but not in the first.

These ambiguities arise because MTR only sees loads and stores, not evictions, which depend on microarchitecture configuration. There are two approaches to resolving these ambiguities. The faster and less accurate approach is to always reconstruct the directory and cache state to reflect the more probable scenario. In the first example described above (Figure 3-10(a)), we mark the block as M, assuming good locality and no communication misses such that the data is not evicted. In the second example (Figure 3-10(b)), we leave *p1* in the sharing vector, assuming that the block remains in *p1*'s cache long enough to be downgraded and written back before being evicted.

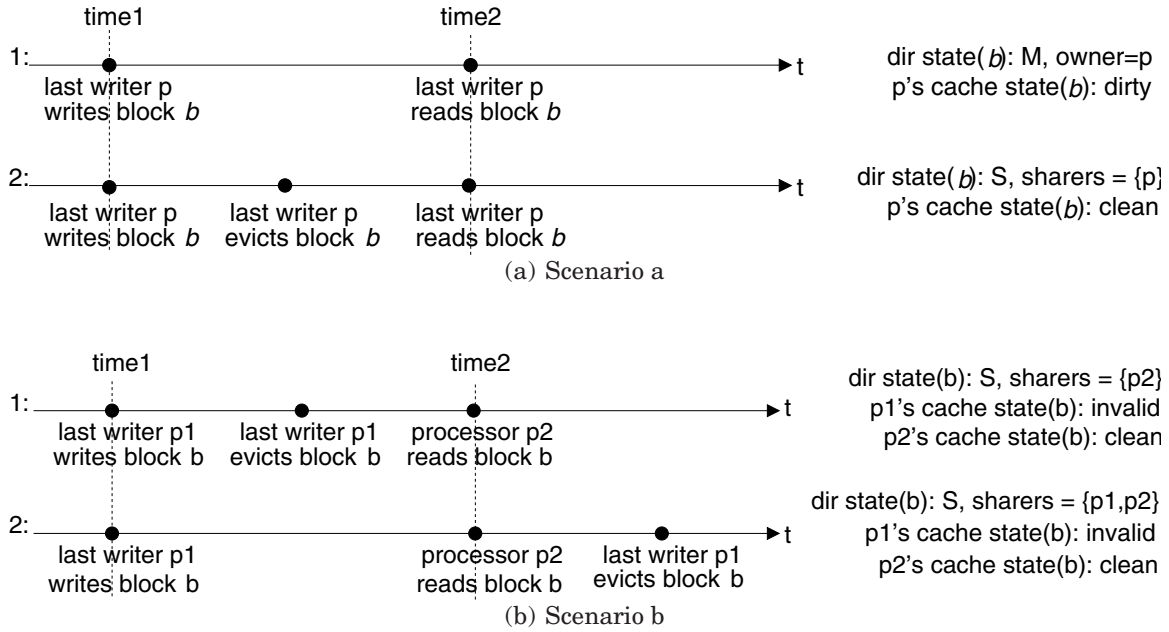


Figure 3-10. Ambiguities from evictions. Scenarios a and b show two alternative sequences of events that result in the MTR holding the same two timestamped accesses at time1 and time2. These ambiguities require additional effort to resolve, though a quick heuristic is usually adequate.

The more accurate approach distinguishes the ambiguous timelines by determining if an owner's copy, installed at *time1*, is evicted before a later access, at *time2*, to the same memory block. A block b is evicted from a set in a k-way associative cache when the processor accesses k new blocks mapping to the same set after it last accesses b. If we re-examine the MTR and find at least k such accesses between *time1* and *time2*, b was definitely evicted in that period. This is shown in Figure 3-11 as Case 1, where there are n accesses to the same set between *time1* and *time2*, and n exceeds the associativity, k.

When the MTR entry contains fewer than k accesses within the time frame (Figure 3-11, Case 2a and 2b), we cannot reach a conclusion about the eviction time — there might have been more memory accesses within that time frame that are not recorded by the MTR. In the figure, Case 2b shows this masking situation: a block cached before *time2* that causes block b to be evicted is accessed again after *time2*, so the previous access is overwritten in the MTR. The MTR contains the same contents in Case 2a and 2b, but Case 2a does not result in a masked eviction. Figure 3-12 shows this algorithm in pseudocode. This special case of irresolvable ambiguity is the tradeoff for being able to maintain so little state in the MTR. However, as we will show in Section 3.3, the unresolved ambiguities are rare, and have little effect on overall accuracy.

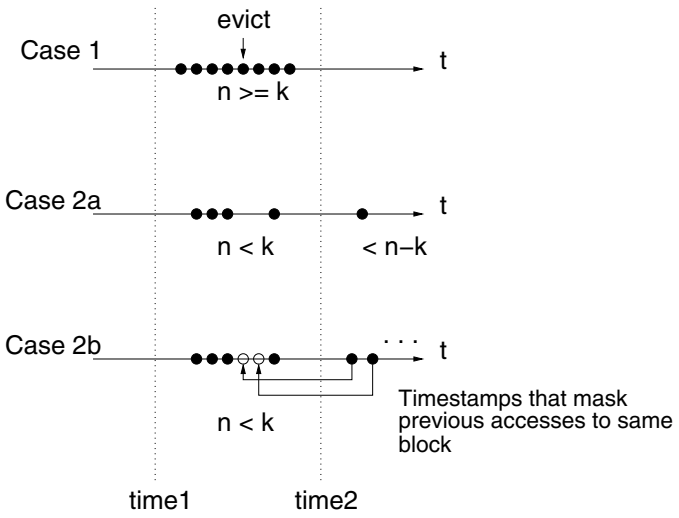


Figure 3-11. Different scenarios when reconstructing evictions. The MTR contains entries marked time1 and time2. The black dots represent accesses recorded in the MTR; the white dots represent accesses no longer visible in the MTR. In the first case, we know for sure that the eviction occurred before the memory access, whereas the second case is ambiguous because of the possibly missing timestamps from the MTR.

3.2 Extending the MTR

In this section, we describe how MTR can be used to reconstruct many different cache configurations and coherence protocols.

3.2.1 Alternative cache configurations

The MTR structure is independent of the simulated cache parameters, with each MTR entry corresponding to a memory block. The state of any sized cache with any associativity can be reconstructed by time sorting touched memory blocks into the appropriate cache sets; cache block sizes that are a multiple of the MTR block size can be easily constructed by merging timestamps from all constituent MTR blocks. Thus, one can use an MTR with word or byte granularity to model fine gradations in block size. In addition, the MTR structure can reconstruct most common forms of time-based replacement policy (e.g., LRU or cache-decay counters). Random replacement approaches LRU for large (> 64 KB) caches [45], so the time-based replacement strategy can be used to closely approximate random replacement.

Multi-level cache hierarchies can be supported by reconstructing the largest cache with MTR and using detailed warming or a further reconstruction for the smaller caches. Inclusive caches may be reconstructed using two independent calls to the cache reconstruction routines, each using different parameters. To reconstruct a non-inclusive cache hierarchy, we first reconstruct the inner caches then omit these cached lines while reconstructing the CSR for the outer caches.

```
IsEvictedBetween(block, cpu, set, k, time1, time2, MTR) {
   count = 0
   ambiguous = 0
   for each entry, i, in MTR that maps to set but is not for block {
      if ((time1 < i.readtime[cpu] < time2) or
          ((time1 < i.writetime < time2) and (cpu == i.writer))) {
         count++
         else if (i.readtime[cpu] > time2) {
           ambiguous++
         }
      }
   }
   if (count > k)
      return true    /* evicted: case 1 of Figure 3-11   */
   else if ((count + ambiguous) > k)
      return false   /* ambiguous: case 2 of Figure 3-11 */
   else
      return false   /* not evicted */
}
```

Figure 3-12. Procedure to determine if *block* is evicted between *time1* and *time2*. If at least *k* other accesses mapping to the same set are found in the time frame, then *block* is evicted before *time2*. If fewer than *k* other accesses are found after *time1*, when *block* is last accessed, *block* has not been evicted. Otherwise, it is unknown when *block* is evicted, and we heuristically assume that the line has not been evicted.

3.2.2 Alternative coherence protocols

MTR can support a variety of directory cache coherence protocols, such as MESI, MOESI, update protocols, and systems with imprecise directory representations. In addition, MTR works with snoopy protocols. Snoopy protocols, in which each participant observes every transaction on a shared bus, are simpler to support than their directory-based counterparts since they only require cache reconstruction.

Explicit clean evictions. An explicit eviction policy notifies the directory about evictions of clean blocks. To support explicit eviction, the directory reconstruction algorithm is modified to keep the directory and cache state consistent. Since the directory should reflect the cache contents exactly, the directory can be built directly from the reconstructed cache state instead of from the MTR. This would improve the reconstruction speed, since we would only have to process every cached block, which may be significantly fewer than every touched block in memory.

MESI invalidation protocol. Here, we illustrate how MTR supports the MESI protocol, which enhances the MSI protocol with an additional exclusive (E) state [84]. A read request for an unshared memory block is cached in the exclusive state as an optimization for read-modify-writes, while a read request for a shared memory block is cached in the shared state

as in the MSI protocol. Under MESI, MTR reconstruction must determine whether a clean line is in the exclusive or shared state. A memory block held by multiple caches must be in the shared state. However, if there is only one cache with a valid and clean copy of a particular memory block, it can be in either exclusive or shared state. The ambiguity is similar to that depicted in Figure 3-10(b). If the directory is notified of P1's dirty writeback, it would grant P2 an exclusive copy of the data, but if P1's copy is not evicted until after P2's read request, P2 would have received a shared copy instead. This ambiguity can usually be resolved using the same technique described earlier for determining eviction times under the MSI protocol.

We have considered, but not evaluated, MTR for use in other invalidate and update-based protocols as described below.

MOESI invalidation protocol. The MOESI protocol adds the owned (O) state to the MESI protocol, which allows writers to directly provide the latest data to subsequent readers without having to write the modified data back to memory [111]. The new readers cache the data in the shared state, while the writer downgrades from the modified to the owned state. The directory records the line as belonging to the writer in the owned state, indicating that the memory has a stale copy of the data. If a processor in the owned state wants to modify the data again, the directory invalidates all of the sharers before upgrading the requesting writer to the modified state. If another processor wants to write the data, all readers are invalidated and the owner must first write the line back to memory.

The main difference between MOESI and MESI is that the last writer of a memory line is downgraded to an owned state rather than a shared state if there are subsequent reads. This translates to the following changes in the reconstruction algorithm:

- The last writer's cached copy is always marked dirty.
- In reconstructing the directory for systems with silent evictions from the MTR, we need to distinguish between the modified and the owned state. If any processor reads the line after the last write, the directory state for that line is owned. Otherwise, the last writer has the line in the modified state.
- In systems with explicit evictions, we reconstruct the directory from the CSR. If one cache has a valid and dirty copy, while other cache(s) have a valid and clean copy of the same memory line, the directory state of the line is owned. If the dirty copy of a memory line is the only copy cached in the system, we need to refer back to the MTR to determine the existence of "missing sharers." In this case, missing sharers refer to processors that have read the same memory line after the last write, but have evicted their copy by the time of reconstruction. This involves a simple scan of the read vector in the corresponding MTR entry to find any timestamps later than the last write timestamp. If there are missing sharers, the line is cached by the last writer in the owned state; otherwise, it is cached in the modified state.

Update protocols. Update protocols, such as the Dragon protocol [71], provide an alternative to invalidate coherence protocols. A processor wishing to modify a shared line becomes the owner of the line, updating the other sharers with the latest value rather than invalidating them. Only one minor change to the MOESI algorithm is needed to support updates: the valid bit is always set during reconstruction, because cached data is always updated instead of invalidated.

Imprecise directory representation. As the number of processors in the system increases, the overhead of maintaining a full bit vector in the directory to represent each sharer becomes exorbitant. Therefore, many multiprocessor systems use a compressed representation of the sharers that records a superset of the actual sharers; these imprecise directory representations increase the invalidation traffic in exchange for reduced directory size. In terms of MTR reconstruction, only the directory representation is changed; the cache state is unaffected.

An example of a system using imprecise directory representation is the SGI Origin [60]. The Origin dynamically switches between the full bit vector and coarse bit vector, depending on where the sharers are located. If all the sharers of a line are in the same physical "octant," the sharers are identified with a full bit vector and an octant number. Otherwise, a coarse bit vector is used, in which every bit of the vector represents a group of processors, and the bit is set if *any* processor in the group is a sharer. To determine the type of directory entry to reconstruct, we examine all the processors that have a timestamp greater than or equal to the last write timestamp. If these processors span more than one octant, the directory has already switched over to the coarse vector mode; otherwise, the directory uses a full bit vector.

3.3 Evaluation

To evaluate the MTR, we employ a flexible and detailed cache-coherent distributed shared memory system model that includes primary caches, main memory with variable latency, and interconnection networks [130]. We drive the memory system with Bochs, a popular x86 full-system SMP-capable emulator [62]. The full-system nature of Bochs (i.e., it boots 4-way SMP Linux 2.4.24) allows us to test the MTR with realistic workloads that require operating system support. Furthermore, the execution-driven nature of the simulator allows our detailed memory system to affect the interleaving of threads, something difficult to achieve in trace-driven simulation.

Several changes are required to the default Bochs distribution. First, a buffer is added to each CPU to track in-progress memory requests. While a request has not yet been fulfilled by the memory system, the CPU's *clock_tick()* function performs no action. Certain x86 instructions which read and modify a value are inherently atomic (e.g., `xchg`) or may include a prefix to force atomic behavior. We patched the simulator to inform the memory system when such an instruction is underway; in these cases, we perform both the read and the write in

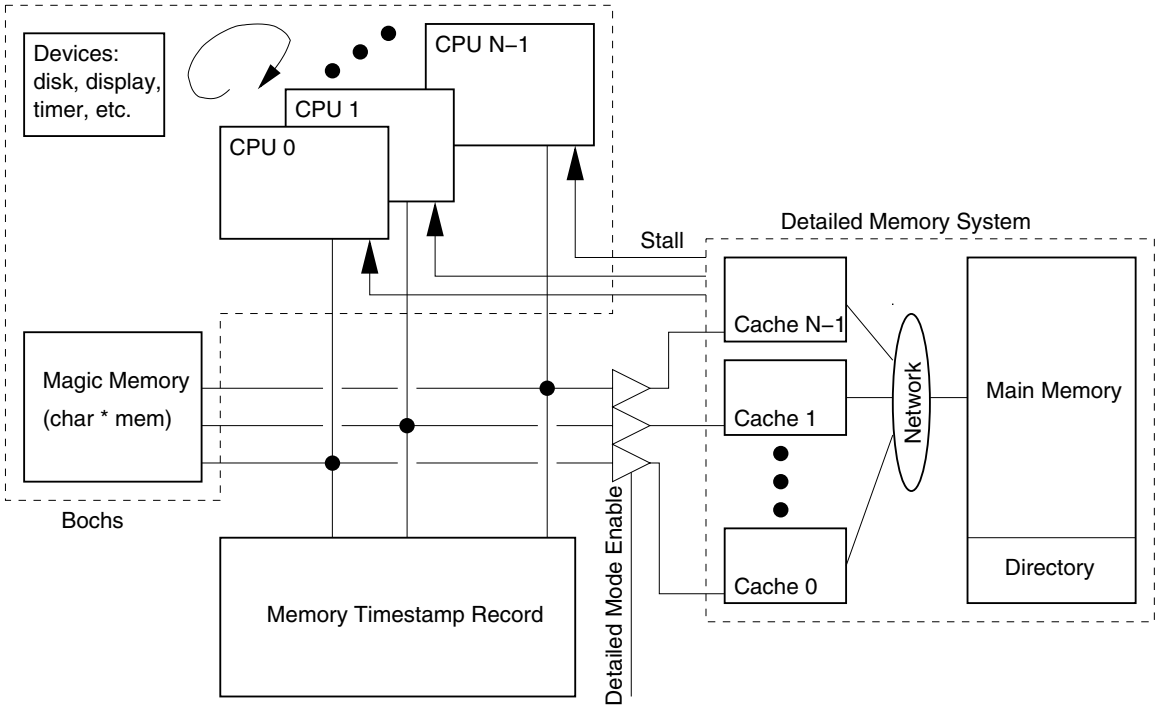


Figure 3-13. Evaluation infrastructure. The detailed memory model stalls a processor's execution based on timing models.

the functional simulator, but issue only the write to our detailed memory system to preserve atomicity. To keep all non-determinism under user control, all simulations begin at a fixed simulated time in the target and Bochs is instructed to disable attempts at resynchronizing the target's realtime clock. Other changes are made to allow batch simulation: updates to the simulated console are logged to a file so we can see output from the simulated application and confirm that it ran successfully. Bochs provides for guest-to-host communication via writes to a special I/O port, and we use this facility to start and stop the simulation.

The overall simulator structure is shown in Figure 3-13. The main loop of the simulator moves round-robin between the CPU models and the devices, incrementing a cycle count at the end of each loop. Those devices which need attention assert an interrupt line and are handled by the operating system on the simulated machine. When fast-forwarding to create an MTR snapshot, the Bochs CPUs access a shared "magic memory" for instant resolution of loads and stores. During detailed simulation, the memory backend performs detailed timing simulation such as cache miss/refill and network routing operations.

We use a simple timing model for this evaluation to allow a greater number of experiments to be completed. We believe the conclusions regarding the MTR's versatility are relatively unaffected by the exact timings chosen, although conclusions about the performance of a particular realistic configuration would obviously require a more detailed system model. We use an in-order processor model that assumes each non-memory instruction takes one cycle to decode and execute. We combine this with a cache of L2-like size and organization,

Number of Processors	4
Cache hit latency	1
Cache organization	4-way, 256 KB
DRAM access latency	20
Cache miss buffer length	16
Network latency	1-cycle to neighbor
Network Topology	2D-mesh

Table 3.1. Simulation Parameters.

but L1-like timing, to approximate the IPC of a modern out-of-order superscalar processor. A low DRAM latency is chosen to shorten simulation time while still causing significant timing-dependent instruction interleaving. Instructions that access the memory system are subject to the latencies of the model (shown in Table 3.1).

The multithreaded workloads used to evaluate MTR include Fortran/OpenMP NAS Parallel Benchmarks [52], server-style benchmarks which spend more time in the OS, and one dynamically scheduled AI benchmark written in Cilk [63]. The NAS Parallel Benchmarks (NPB) are representative kernels from the area of computational fluid dynamics. Understanding their computation requires very specific mathematical knowledge, so we treat the applications opaquely: as a self-verifying set of programs that contains data-level parallelism in do-all loops and exhibits various useful communications patterns. Refer to Table 3.2 for a description of the benchmarks.[3]

For simulation automation, benchmarks were invoked in a runlevel without superfluous processes/daemons to ensure that non-essential processes do not interfere with the benchmark. Each benchmark's inputs were chosen to allow detailed runs to complete within twelve hours on a 2.2 GHz Pentium 4.

3.3.1 Effect of inherent MTR inaccuracies

When sampling with the MTR, there are several conditions that can make it difficult to estimate a metric that will be seen in full detailed simulation. Sampling itself provides only an estimate of performance because not all instructions are simulated in detailed. Additional error can develop due to lack of information in the MTR forcing the reconstruction algorithms to use a heuristic to guess the status of a line. Another cause of error is a change in memory access interleaving brought about by extended periods of functional simulation. This access order, caused by instant resolution of loads and stores by the functional simulator, is not representative of a target with latency, and it can allow a benchmark to experience thread interleavings unlikely to occur in a hardware implementation. This is similar to the concept of *space variability* [5].

[3]The IS (integer sort) component of NPB is not always included due to difficulty verifying its output when using custom input parameters. When we revert to the "class W" input, IS runs correctly and appears with the results in Section 3.3.1.

Benchmark	Description
BT	NPB: block-tridiagonal CFD application, class S
CG	NPB: conjugate gradient kernel, class S
EP	NPB: embarrassingly parallel kernel, class W
FT	NPB: 3X 1D Fast Fourier Transform, class S (-O0)
LU	NPB: lower-upper decomposition with SSOR CFD application, class S
MG	NPB: multigrid kernel, class W
SP	NPB: scalar pentagonal CFD application, class S
dbench	executes Samba-like file-oriented system calls 3 clients, 10000 requests (gcc 2.96)
apache	Apache web server benchmark "ab" worker threading model, 2000 requests, 3 at a time (gcc 2.96)
ck	Cilk checkers (parallel alpha-beta search) 4 processors, black plies 6, white plies 5 (Cilk 5.3.2,gcc 2.96)

Table 3.2. Benchmark Description. All benchmarks are compiled with the Intel Fortran Compiler version 8 using the options "-g -O2 -openmp" unless noted.

Table 3.3 shows the effect of space variability without the error introduced by MTR ambiguities. Two detailed simulations, non-blocking and blocking, are run without the MTR. For the blocking simulation, we changed the scheduling algorithm of our SMP simulator: instead of allowing un-blocked processors to continue while a different processor waits for memory, we instituted a blocking schedule in which one processor's memory request is satisfied before another processor is allowed to continue. This strategy produces incorrect performance numbers, but causes the order of memory accesses to match that of functional simulation, which also satisfies a processor's request for memory immediately and before another processor issues a request. While we use the same suite of benchmarks described earlier, the results in this section are run on an eight-CPU target with a four-way, 16 KB cache per processor. There are three cycles of network latency between processors and a miss penalty of approximately 112 cycles. The resulting cache miss rates are shown in the center columns. The third column is the difference relative to the non-blocking schedule. Five of the eleven benchmarks have a magnitude of difference greater than 10%. The table confirms that space variability is a challenge, even without the inclusion of the MTR's inherent errors.

To we examine the MTR error independent from the space variability error, we use the MTR and the blocking schedule described above during detailed simulation. Table 3.4 shows the error in the MTR's miss rate prediction when compared to two baselines. The first baseline shows each benchmark in its entirety using our default detailed simulator, allowing non-blocked processors to continue while other processors wait for memory. Because the bulk of a sampled simulation is spent with an ideal memory model, applications that exhibit space variability are difficult to estimate with the MTR as we see in the column labeled "MTR Miss rate prediction difference (%) vs. non-blocking detailed sim.".

Next, we conduct a blocking detailed simulation in which we force all processors to block when any processor is waiting for memory. This causes the processors to order memory re-

Benchmark	Miss rate		Relative Difference (%)
	Non-Blocking	Blocking	
ap	0.0907	0.0910	0.26
bt	0.0152	0.0166	9.22
cg	0.0510	0.0540	5.97
ck	0.0118	0.0100	-15.63
db	0.0468	0.0435	-7.01
ep	0.0158	0.0160	1.37
ft	0.0607	0.0644	6.05
is	0.0253	0.0337	33.30
lu	0.0067	0.0314	370.77
mg	0.0122	0.0300	146.02
sp	0.0308	0.0137	-55.50

Table 3.3. Difference in miss rate caused by memory access ordering can be substantial.

Benchmark	MTR miss rate estimate	MTR miss rate prediction difference (%)	
		vs. non-blocking detailed sim.	vs. blocking detailed sim.
ap	0.0834	-8.1130	-8.3492
bt	0.0161	6.2156	-2.7519
cg	0.0538	5.5250	-0.4226
ck	0.0095	-19.5555	-4.6573
db	0.0410	-12.4149	-5.8153
ep	0.0159	0.6949	-0.6642
ft	0.0610	0.5438	-5.1932
is	0.0327	29.2144	-3.0624
lu	0.0293	339.5416	-6.6335
mg	0.0263	115.7782	-12.2907
sp	0.0115	-62.5971	-15.9399

Table 3.4. The MTR is affected by multiple error sources. Even when space variability is removed, sampling and MTR-induced ambiguities introduce error.

benchmark	time (sec)		difference (sec)	slowdown (%)
	default (M)	resolve (D)		
ap	1754	1759	5	0.28
bt	666	657	-10	-1.44
cg	1423	1402	-21	-1.46
ck	14035	14098	63	0.45
db	4293	4262	-31	-0.73
ep	2316	2017	-299	-12.90
ft	1016	996	-20	-1.93
is	479	491	12	2.53
lu	330	321	-9	-2.62
mg	1113	1111	-3	-0.24
sp	302	300	-3	-0.83

Table 3.5. Slowdown incurred by ambiguity resolution.

quests in the same way as a functional simulator. The rightmost column of the table compares the MTR miss rate prediction to that of the blocking simulator. The MTR's error compared to the blocking simulator is generally lower (FT is an exception) because the memory ordering of the blocking detailed simulation matches that of the MTR-based simulation. The remaining difference between the blocking simulation and the MTR is mostly due to sampling and ambiguities during MTR reconstruction.

3.3.2 Slowdown due to ambiguity resolution

The `IsEvictedBetween` procedure provides information that is useful during reconstruction. We now evaluate the time needed to perform the procedure. Using a sample size of 128 K instructions and a sampling ratio of 1:100, we run each benchmark to completion. To reduce the effect of other processes, we request exclusive use of batch servers, each with identical configurations. We measure the runtime using the sum of user and system seconds reported by the C library's `getrusage()` function. Each benchmark is run three times, and we show the average of the three.

Table 3.5 shows that the cost of ambiguity resolution is less than 3%. Often, the default policy actually requires more time. It is difficult to determine whether this is the result of noise on the host causing timing variation, or whether the alternate orderings induced by the M policy cause the slowdown.

3.3.3 Online sampling performance

In this section we examine the accuracy and speed of online sampling with the MTR. To perform online sampling, the detailed simulator's cache and directory are reconstructed from the MTR at every transition from functional to detailed simulation. When transitioning back to functional simulation, the processors halt execution until all of the outstanding memory requests are serviced. To reduce the chance of coinciding with a periodic behavior in the

1. Generate a pseudorandom integer, $0 < k \leq 2r$
2. Enable fast simulation; disable statistics
3. Run ki instructions
4. Enable detailed timing simulation
5. Run w instructions to warm up network queues and memory system buffers
6. Enable statistics
7. Run i instructions
8. Repeat

Figure 3-14. Online sampling methodology. Parameters: warm-up duration (w), sample size (i), and sampling ratio (r).

benchmark, we randomly sample the detailed portions rather than rely on periodic sampling across the duration of the benchmark. We choose a warm-up duration (w), sample size (i), and a sampling ratio (r), and simulation proceeds according to the steps in Figure 3-14.[4]

To provide a baseline, we added a fast functional warming (FFW) mode to our simulator, which is a straightforward SMP extension of earlier uniprocessor functional warming work [127]. FFW fast-forwards the simulation by updating the cache and directory state in each simulated cycle, but these FFW updates are significantly more expensive than MTR updates. For each memory request during fast-forwarding, FFW must first calculate a set index, then search all ways of the local cache to perform a tag check. If the request results in a hit, FFW must update the local LRU information; otherwise, multiple non-local caches and the directory must be updated to reflect all of the downgrades or invalidations caused by the cache refill. Whereas each MTR update runs in constant time, FFW updates scale with the number of caches and associativity, and can vary due to miss rates and sharing patterns. FFW is also more difficult to parallelize than MTR, requiring some form of mutual exclusion to implement parallel cache and directory state updates correctly. On the other hand, FFW does not require any form of reconstruction during the transition between fast to detailed mode, because the directory and cache state is kept up-to-date during fast-forwarding.

Fast-forwarded simulation yields vastly different thread interleavings than detailed simulation, since processors do not stall on memory instructions during fast-forwarding. It is well known that even small changes in SMP system timing often introduce large variations in simulation results [5]. It is therefore meaningless to compare a single run of detailed simulation and a single run of fast-forwarded simulation, because the different results may

[4]Another potential method for choosing sampling ratios would be to randomly skip $9i$–$11i$ (for a 1:10 average ratio) and $92i$–$108i$ (for 1:100). These narrower choices increase error for all but LU and SP, the two benchmarks with the fewest samples.

simply reflect the variation introduced by different, but still representative, thread interleavings rather than differences in simulation accuracy. We present the fast-forwarding results in the context of timing variations induced by altering system parameters.

We introduce two sources of variability in our system. First, we enable Bochs's slowdown timer component, which keeps the emulator in sync with real time on the host, and causes emulated devices to be handled at nondeterministic rates, varying the OS scheduling of threads. Second, we model changing processor workloads by choosing a different processor to run 25% slower than its peers every 10,000 instructions. Repeated simulation runs with these timing variations capture various representative thread interleavings and coherence race conditions. We use 100,000-instruction measurement samples, which should be long enough to span the duration of practical coherence races on our target, so samples (of which we take hundreds per run) should observe races in proportion to their occurrence in full runs.

Accuracy comparison. We compares the cache miss rate and coherence message count (under the MSI coherence protocol) reported by detailed simulation, FFW, and MTR. Figure 3-15 shows these metrics for only one cache, but the results are similar for the other caches. We use *FFW(a:b)* and *MTR(a:b)* to denote the results of FFW and MTR where a:b is the ratio of instructions executed in the detailed period to those executed in the fast period. There are seven bars for each reported metric, each representing a simulation configuration: fully detailed run, FFW(1:10), FFW(1:100), FFW(1:1000), MTR(1:10), MTR(1:100), and MTR(1:1000).

We also remind the reader that the detailed runs can only capture *some* of the legal thread interleavings, which may differ significantly from those captured by their corresponding fast-forwarding runs. This means it is meaningless to state an error compared to detailed simulation because there are many potential detailed simulations; rather, we would like to see the confidence intervals of a sampled run fall within those of a detailed run. In this case, there are two types of samples: 1) detailed samples within a benchmark and 2) each complete benchmark run using different timing variation. It is prohibitive to perform enough runs of a single benchmark to establish tight confidence intervals, so we report each result as median of eight separate runs (denoted by the solid bars), and the range of values observed (the thin, I-shaped lines). We do not observe, and there is no reason to expect, a normal distribution of miss rates or message counts across runs.

In an attempt to quantify error incurred by our techniques, we note the largest excursion from the detailed run's median, normalized to the detailed median. Most accelerated benchmarks, with a detailed to fast ratio of 1:10, have an error in miss rate of less than 15%. LU, SP, and Apache exceed this error. For LU and SP, our shortest benchmarks, the problem is likely due to an inadequate number of detailed samples. Both MTR and FFW exhibit similar deviations from the detailed simulations — an effect most prominent in Apache. The technique we use to introduce variability by selectively adding processor stalls has a much larger relative impact on the fast-forwarding schemes than on the fully detailed run, where

Benchmark	Instructions (Millions)	Mem Refs (Millions)	Ambiguous Addresses	Touched blocks / mem refs (%)
BT	780	400	95632	0.23
CG	792	390	111818	0.42
EP	4306	2076	93786	1.40
FT	5264	2964	106356	0.44
LU	368	172	105995	0.24
MG	2621	1549	292698	1.78
SP	361	164	101855	0.23
dbench	2692	867	180328	1.07
apache	2782	947	143332	0.27
ck	2617	396	102389	0.12

Table 3.6. Benchmark characteristics: Instructions and Memory Reference columns reflect full detailed runs of the benchmark. Ambiguity and touched block stats are for 1:100 MTR runs with 100,000-instruction samples.

processor CPI is already much higher due to memory system stalls. We believe this explains the bias and generally greater variance observed with the fast-forwarding schemes versus the detailed model.

Increasing the ratio to 1:1000 leads to an unacceptable fast forwarding error in most cases. We achieve a good balance of error and speed when we set the detailed to fast ratio to 1:100. At this rate, six of the ten benchmarks have error within 25%, with the best performers being EP, MG, and CG with error below 12%. These benchmarks have relatively fewer invalidation and downgrade requests, indicating simpler sharing behaviors that are less likely to be perturbed by the effects of fast-forwarding.

Despite using different fast-forwarding strategies, both FFW and MTR report similar results for all metrics. The slight discrepancies can usually be attributed to the MTR's assumption that all ambiguous blocks should be marked "modified." Table 3.6 lists the number of ambiguously reconstructed addresses. The number of ambiguities does not always correlate directly with the accuracy of MTR fast-forwarding. First, although the number of ambiguities may be high, the resulting error may be small if we always pick the correct way to reconstruct. Second, the ambiguously reconstructed blocks may not be needed again, which is likely in applications with low temporal locality. Third, the error caused by incorrect reconstruction may be negligible compared to the simulation's variability, so it does not affect the overall accuracy of the fast-forwarding simulation. To reduce error further, we can adopt the more sophisticated approach of establishing eviction times to help resolve ambiguities at the cost of slower reconstruction as described below.

Case study. The ultimate goal of sampling is to allow designers to make quick and reliable architectural. This section presents a simple example to show that results obtained by online sampling during multiprocessor simulations can provide as much insight as those from de-

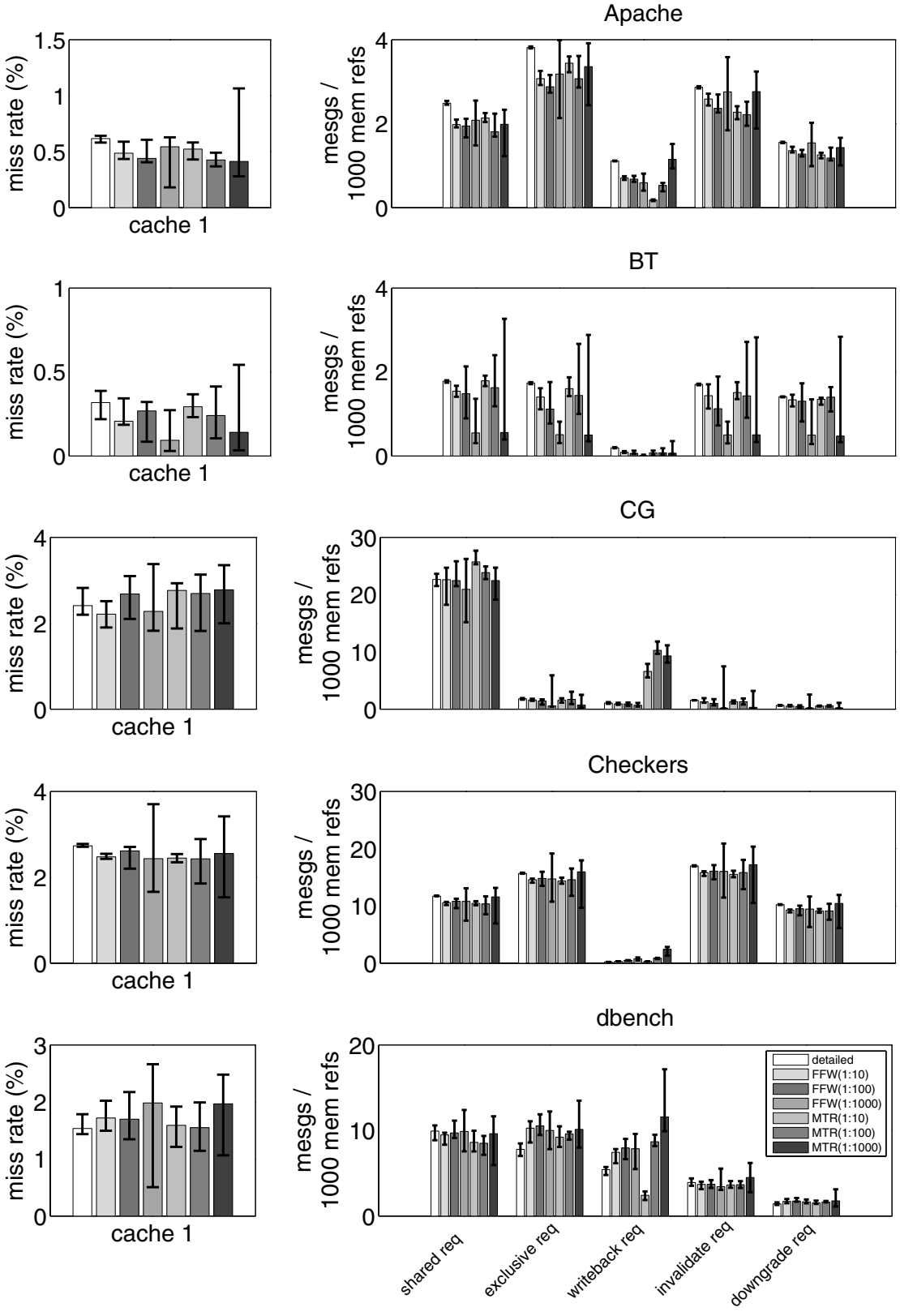


Figure 3-15. Accuracy comparison of FFW and MTR to detailed simulation. Each bar represents the median of eight individual runs of the benchmark. Max and min are noted with thin lines. No ambiguity resolution is performed.

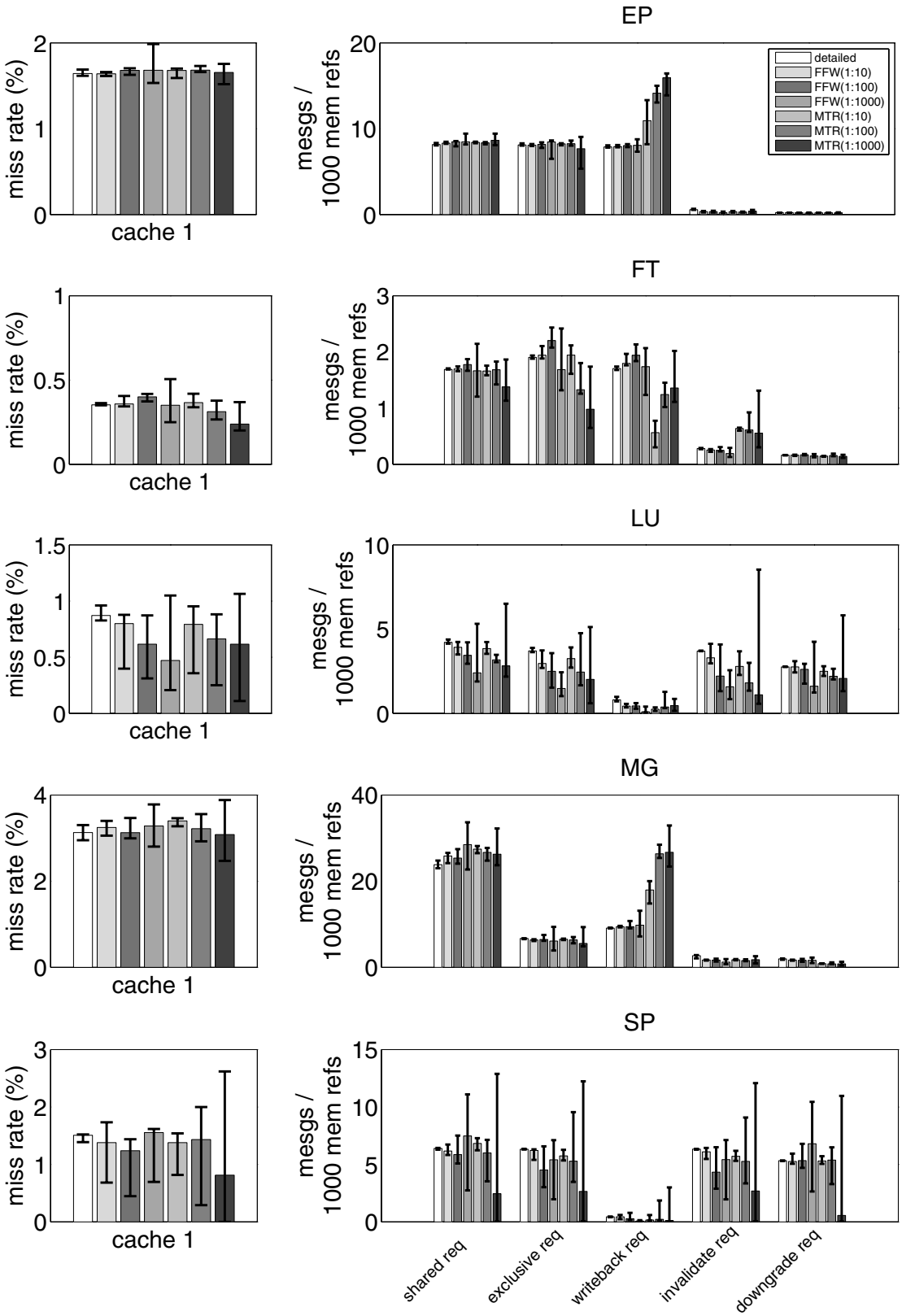


Figure 3-15 (continued)

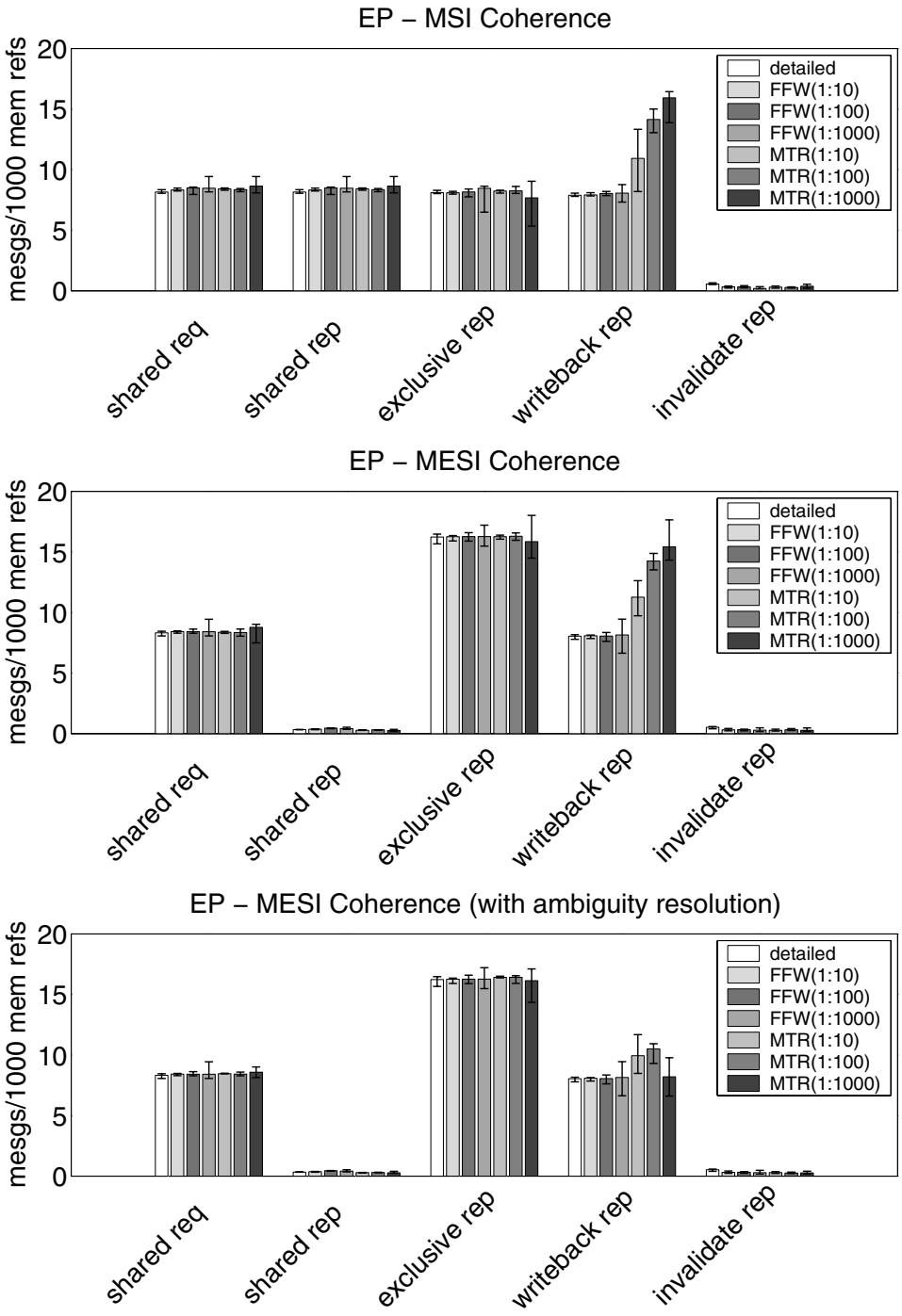


Figure 3-16. Comparison of MSI and MESI protocols. The bottom figure shows the reduction in writeback messages caused by resolving M/S ambiguities as described in Section 3.1.5.

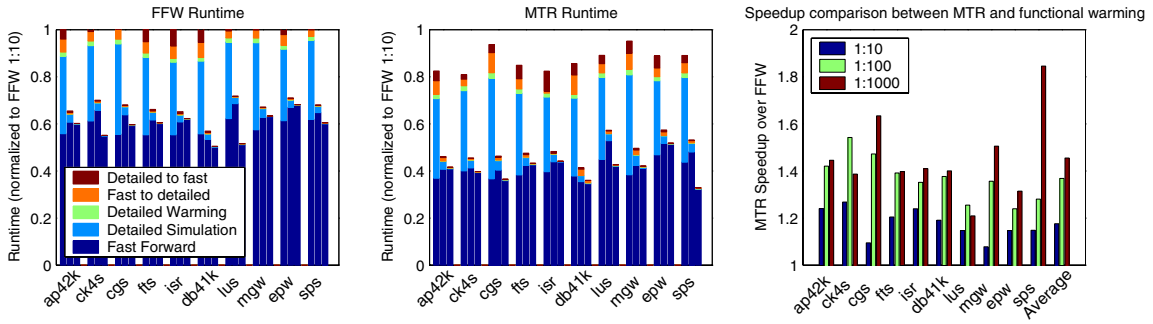


Figure 3-17. Normalized running time of FFW and MTR and the relative speedup of MTR over FFW. Each group of three bars represents the run time with detailed-to-fast ratios of 1:10, 1:100, and 1:1000. Each bar separately shows time spent in the three phases of simulation including transitions.

tailed runs. We have run the EP benchmark under both the MESI and the MSI protocols, and the collected coherence message metrics are shown in Figure 3-16. As expected (unless many read-modify-writes are present), shared requests under MESI and MSI are similar. However, the shared *replies* are quite different, since part of the shared reply messages in MSI become exclusive reply messages in the MESI protocol. The reduction in shared replies with MESI is prominent in both sampled and complete detailed simulation data, thus allowing one to draw the same conclusion about the effects of MESI versus MSI with much shorter simulations.

Despite indicating a much larger absolute number of writeback replies than detailed or FFW runs, MTR runs present similar writeback replies under both protocols, which allows one to draw the correct conclusion about the effect of MESI on writeback replies. This MTR discrepancy in writeback replies is due to our assumption that certain ambiguous blocks should be marked M (modified) rather than S (shared). When we enhance MTR reconstruction to resolve this category of ambiguities, MTR results approach their detailed and FFW counterparts, because fewer spurious writeback messages get generated by incorrectly marked M blocks (bottom graph of Figure 3-16).

Speedup. In online sampling, snapshot generation time is a major component of overall runtime. Figure 3-17 compares the running times of our two online sampling schemes. Each group of three bars represents the benchmark's execution time normalized to the slowest run for that benchmark. The three bars represent detailed-to-fast ratios of 1:10, 1:100, and 1:1000 respectively. When only 1% or fewer instructions are run in detailed mode, over 95% of the simulation time is spent in fast mode. The small fast-to-detailed transition times confirm that the reconstruction time of the MTR scheme does not outweigh the speedup it can provide during fast mode. This is largely due to the fact that while our programs can make billions of memory references, the MTR compresses repeated references to the same block. Table 3.6 shows the total number of memory references during all of execution and the ratio of "touched blocks" to total references. The number of touched blocks which must be

$$C = \frac{(T_{fast,FFW} + T_{slow,FFW}) - (T_{fast,MTR} + T_{slow,MTR})}{T_{slow,MTR}} + 1$$

Figure 3-18. Determining number of configurations possible in previously allotted time. In this equation, C represents number of configurations, $T_{speed,scheme}$ is an absolute time, and T_{slow} includes all reconstruction time.

considered during reconstruction is a small fraction of the total requests — usually less than half a percent.

MTR edges out FFW by up to 1.45× speedup on average, as shown in Figure 3-17. While MTR is always faster than FFW, the relative improvement due to sampling ratio is effected by variation, number of touched blocks, and particular sharing scenarios. Although MTR achieves a respectable speedup in the serial execution of a single configuration, what is more exciting is the additional speedup from multi-configuration simulation and parallelization. Using the equation in Figure 3-18 and assuming 1% detailed execution time, we estimate that MTR could simulate five different configurations simultaneously and still have comparable running time to FFW running one configuration. The numerator represents the amount of time saved by using MTR instead of FFW. This time can be spent reconstructing and executing additional detailed simulations — each requiring $T_{slow,MTR}$.

For completeness, we note that the MTR is 7.7 times faster than our detailed model when using the relatively accurate 1:100 sampling ratio. FFW is 5.5 times faster at this ratio. Of course, much higher speedups will be achieved when using a more complex detailed model, for example, with a detailed DRAM model in place of our fixed-latency memory, or an out-of-order superscalar processor model instead of our in-order single-issue processor model.

3.3.4 Checkpointing with MTR

Checkpointing removes the overhead of repeatedly creating MTRs with functional simulation when evaluating multiple microarchitectural configurations. However, it is more difficult to implement and requires large files on disk to hold the memory contents at each checkpoint. MTR snapshots can be added to architectural checkpoints to reduce the amount of detailed warming required. Because the MTR is microarchitecture-independent, many different configurations can be initialized from the single snapshot.

The MTR representation is highly compressible. A significant savings is achieved in the spatial dimension using the multilevel valid-bit scheme, discussed in Section 3.1.4. This eliminates the need to store addresses along with data: the address can be reconstructed based on offset within the snapshot and the empty page bit vector. In the temporal dimension, delta encoding may be used instead of full timestamps — although we show that this offers relatively minor benefit for short warming periods.

Scheme ID
Number of blocks
Page size
Block size
Number of CPUs (N)

Table 3.7. Header of compressed MTR.

To examine the extent to which the MTR can be compressed, we implemented several compression schemes. Each scheme uses a common header that contains the fields shown in Table 3.7. The Scheme ID indicates which format is in use and allows proper interpretation of the structure. The number of blocks can be divided by the page size to discover the number of pages; several schemes require this number to parse the next block of the snapshot. Block size is used to infer the address represented by each MTR entry, and the number of CPUs is needed to determine the entry's size.

The schemes are enumerated below:

1. **All.** Each block of the MTR is written to disk using the variable-length representation shown in Table 3.8. We assume 1 byte to store the CPU identifier (supporting up to 256 CPUs) and 4 bytes for a logical timestamp (supporting up to 2^{32} memory accesses in a warming period). Output is written at a byte-granularity as it is more easily compressed by general purpose compressors such as gzip and bzip2.

2. **Accessed.** The output begins with a vector denoting which pages of the MTR contain blocks that have been accessed. Only those pages indicated by the vector are written using the format of the *All* scheme.

3. **Accessed unshared.** Even in programs that share memory among several processors, many memory blocks are accessed by just a single processor. This scheme employs the alternate encoding shown in Table 3.9 to reduce the space occupied by these common single-access cases.

4. **Accessed horizontal.** We begin with the vector of accessed pages described above. When outputting MTR records, write time is emitted explicitly, but read times are reported as deltas from the write time. The hope is that the read timestamp arrays become similar arrays of small numbers which would be easier to encode than monotonically growing timestamps.

5. **Accessed vertical.** We begin with the accessed page vector. The first valid record on each page is denoted the *base*. Additional write times on the same page are reported using a delta from the base write time. Additional reads are reported as deltas from the N base read times. This scheme should work well in the presence of locality at the page granularity.

	Valid	Remainder	Record Size (Bytes)
Valid entries	1	< writer, last write time, read time array >	1+(1+4+4N)
Invalid entries	0	<>	1

Table 3.8. Variable length MTR records: default encoding.

	Code	Contents	Record Size (bytes)
Invalid entries	0	<>	1
Single writer	1	< writer, last write time >	6
Single reader	2	< reader, last read time >	6
Default	3	< writer, last write time, read time array >	6+4N

Table 3.9. Variable length MTR records: *Accessed unshared* encoding.

6. **Accessed vertical2.** Same as *Accessed vertical*. However, after each block is emitted, its times become the new base time so that each block's times are output relative to the previous block on the page. This should work well for unit-stride accesses.

7. **Accessed both.** This scheme combines the horizontal (temporal) and vertical (spatial) approaches. We record a base write time for the first valid block on a page. Read times for the first block are reported relative to the base write time. Write times for following blocks are also reported relative to the base write time. Read times for following blocks are reported as the difference between the actual read time and difference between the record's write time and the base write time:
delta time = mtr[record].readtime[cpu] - base write time - mtr[record].writetime.

We use two baselines, similar to the analysis in Section 4.4.1. The first is a trace of all memory accesses. Like the MTR, such a representation allows reconstruction of any cache or directory state, but unlike the MTR, the trace is likely to contain many references that are not actually necessary to warm-up state. Each reference is represented with the fields shown in Table 3.10. Second, we consider a direct serialization of the contents of each CPU's cache. Storing concrete cache state allows quick reconstruction, but only for the microarchitecture that has been stored. We assume a 4-way, 16 KB cache per CPU.

The cache state is serialized to a snapshot using a method similar to the MTR *All* scheme as shown in Table 3.11. The encoded cache blocks are preceded by a header which provides the number of blocks, block size, and the number of bytes per way. A footer provides the LRU metadata for each set. Each of the three representations (MTR, Memory Trace, and Concrete Cache) is compressed with gzip and bzip2. The per-cpu cache snapshots are combined with

State	Size (Bytes)
Load/Store	1
CPU	1
Address	4

Table 3.10. Format of memory reference trace.

	Valid	Remainder	Record Size (Bytes)
Valid entries	1	< tag, dirty >	6
Invalid entries	0	<>	1

Table 3.11. Variable length encoding of cache blocks.

`tar` before compression so that the general-purpose compressor is able to take advantage of similarities between individual snapshots.

Each benchmark is run until it has completed one unit of work, advancing to each sample in an online mode. State snapshots are taken at the end of each warming period. The MTR is cleared, and the process repeats. Thus, the nth snapshot, snap_n contains information about the memory accesses that have occurred between samples $n-1$ and n. Reconstructing the state at sample n requires snap_1 through snap_n inclusive.

Figures 3-19 and 3-20 show compression ratio normalized to the size required to store a snapshot of eight 16 KB caches (compressed file size / compresses cache snapshot size). The bars are annotated with compression ratio when it is off the scale. The figures show that the compressed MTR always requires less storage than the memory trace. In several cases, the memory trace does surprisingly well when compared to the MTR and would likely improve if specialized compression were applied. However, if larger samples are desired, the self-compressing nature of the MTR is an advantage. Furthermore, while a trace is versatile, it requires time to simulate each of the memory accesses to warm a cache. MTR reconstruction, however, scales with the number of entries in the MTR.

With gzip compression, the MTR requires 6.6–7.1 times more storage than eight 16 KB caches (depending on the length of the fast-forward period), but is more flexible than the small concrete cache snapshot. With bzip2 compression, the MTR improves, requiring 6.3–6.9 times the space of a concrete snapshot. Memory traces require 55–65 times more space than the cache snapshot, or 25–31 times the space when bzip2 is used. Interestingly, bzip2 is able to make the MTR's *All* format compress to almost the same size as its *Accessed* format.

The simple *Accessed unshared* scheme proves almost as good as the techniques which use time deltas. This is likely due to the fact that, during relatively short warm-up periods, regular time strides are not nearly as common as singly-accessed blocks. Though the use of deltas reduces the number of bits that are needed to represent a timestamp, deltas do not skew the distribution enough to aid the entropy coding of gzip and bzip2. For example a bunch of large numbers such as "36114" can be encoded with just as few bits as an equally likely bunch of small numbers like "14."

The amount of storage can be dramatically reduced — by a factor of 720 — by storing only memory access information for those blocks that will be accessed during an ensuing detailed sample point [125]. This requires that the simulation progress through the instructions belonging to the detailed sample, record the required memory addresses, and include only information about these addresses in the preceding snapshot. Such a method excludes

references generated by incorrect speculation. In practice, few such accesses are observed, so the impact on performance estimation is small. This method of excluding portions of memory was proposed to reduce the amount of memory *values* needed in a memory snapshot, but could similarly be applied to exclude the corresponding unneeded memory *metadata* from an MTR snapshot.

3.4 Interfacing with the MTR

To cleanly integrate an existing multiprocessor cache simulator with the MTR requires the following methods:

```
get_blocksize()
get_cachesize()
get_ways()
store(int cpu_id, addr_t addr, eviction_t & evictions);
load(int cpu_id, addr_t addr, eviction_t & evictions);
invalidate(int cpu_id, addr_t addr, eviction_t & evictions);
```

The goal of the interface is to expose as few details about a given cache simulator to the MTR as possible. Rather than a block-by-block reconstruction, the interface supports an indirect means of reconstructing the state of the detailed simulator's cache. This decoupling allows the MTR and cache simulator to be modified and improved without affecting each other. In addition, reconstruction automatically respects the replacement policy of the cache without changes to the reconstruction algorithms.

`eviction_t` is a structure that contains the identity of the evicting CPU and the address of an evicted cache block. The `load()` and `store()` methods are used to restore caches from the CSR. The methods have ordinary cache semantics: e.g., `store(n, addr, evictions);` informs the cache that the nth CPU is performing a write to address `addr` (the data can also be sent if required by the simulator). In addition to performing the load or store and the appropriate actions of the cache (tag checks, replacement, writeback, etc.), the `load()` and `store()` methods fill in the `eviction` reference argument with the block (if any) that is evicted as a result of the call. This `eviction` argument informs the MTR of evicted lines. The MTR uses this information when reconstructing the directory to prevent lines that have been dropped from the cache from erroneously being labeled as shared.

The directory interface must include the functions `set_sharers(int addr, vector <int> sharers)` and `set_state(int addr, state_t state)`, which are called by the MTR to reconstruct the directory. `set_state` sets the state of the memory block at address `addr` to `state` (e.g., Modified). `set_sharers` sets the sharing vector of a block, or specifies its sole owner if `sharers` has just one element.

Special steps are taken to support alternating between slow and fast mode, merging newly reconstructed cache state with the state already present. When the CSR contains *ways*

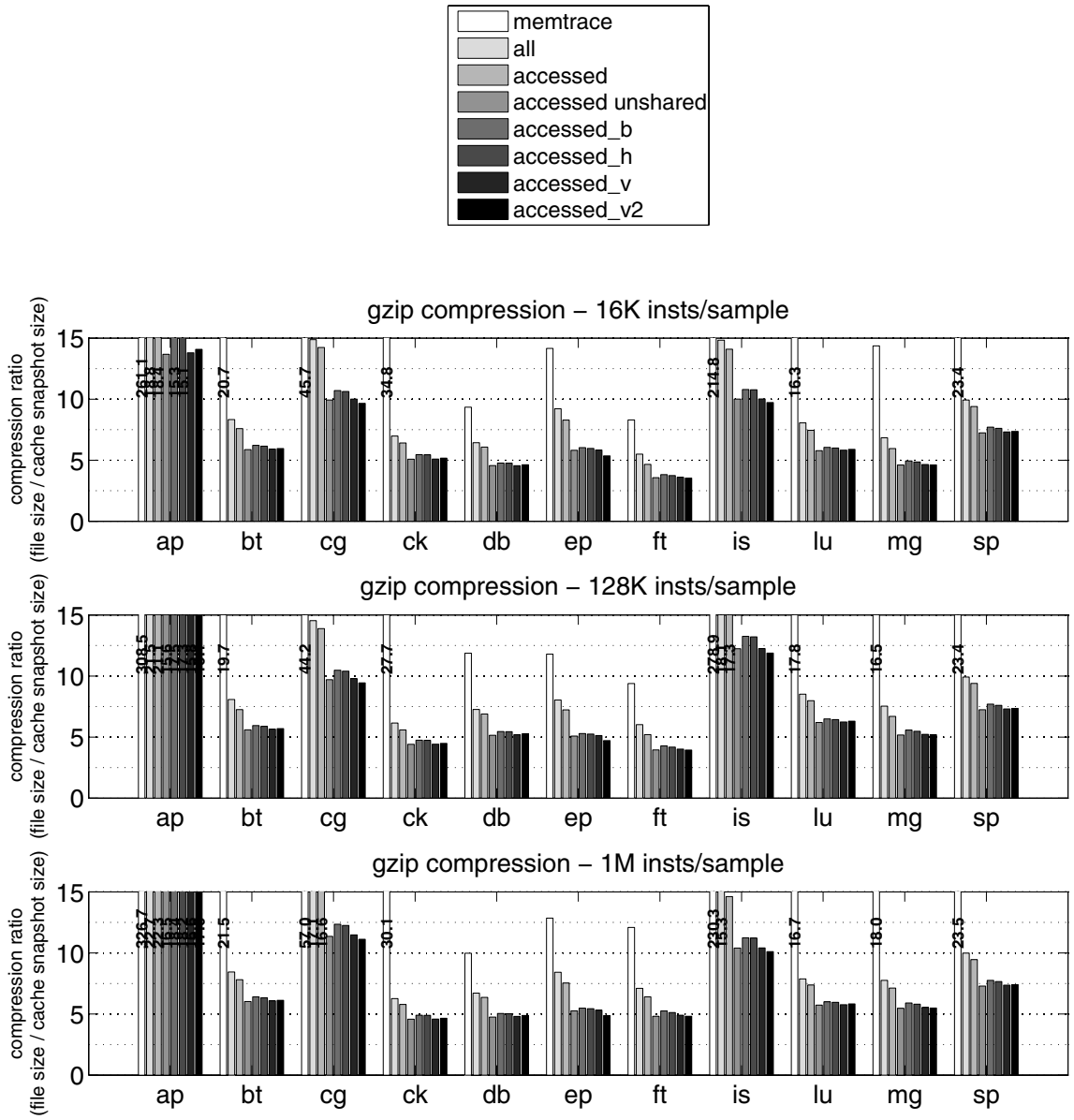


Figure 3-19. The MTR is easily compressed (gzip).

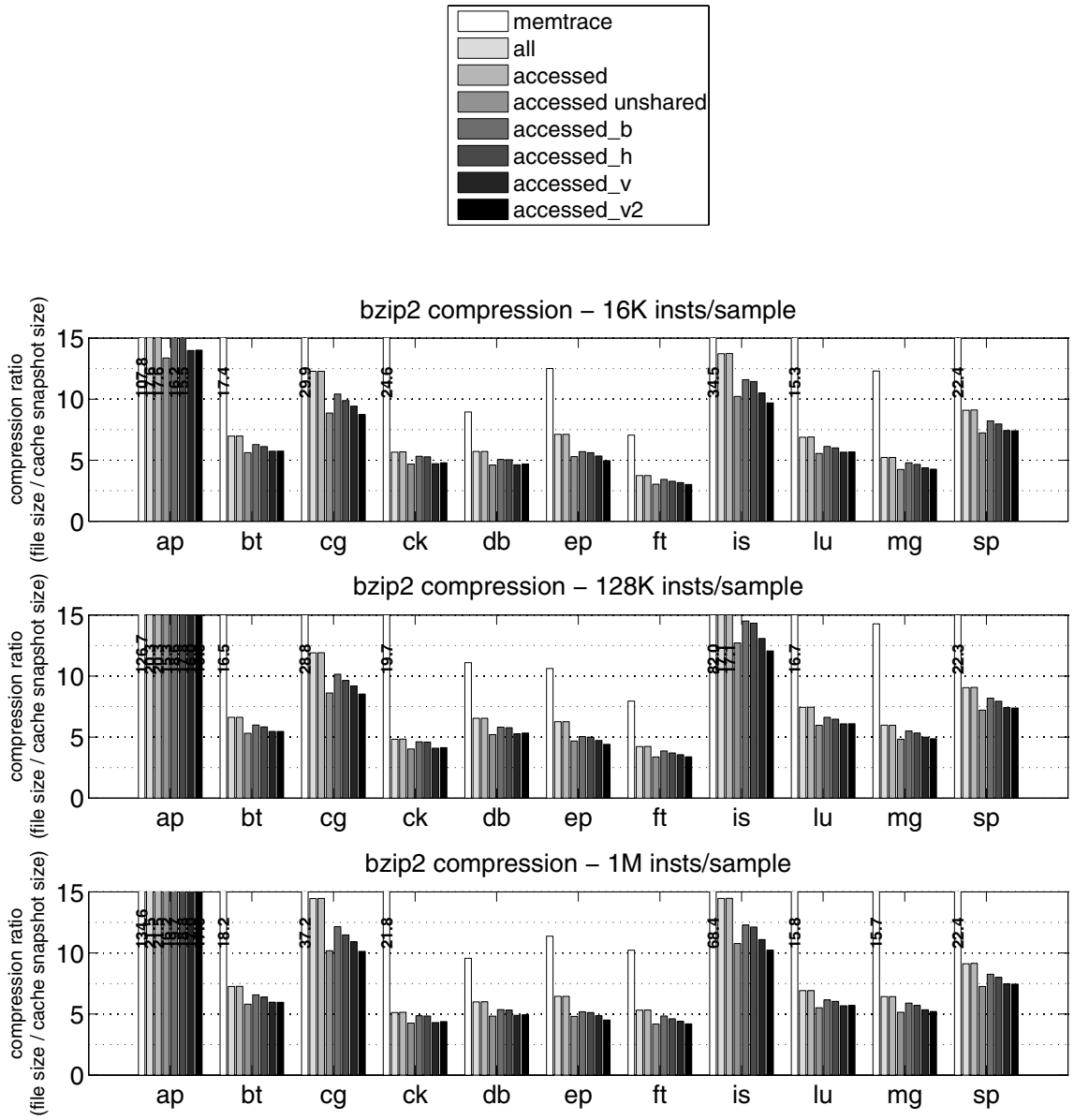


Figure 3-20. The MTR is easily compressed (bzip2).

elements of a set, there is a possibility that some number of memory references has occurred above and beyond *ways*. Were we to merely reconstruct the *ways* blocks that we know about, a dirty block already present in the cache could cause a `load()`ed line to appear dirty when the CSR knows that it is clean. In this case, we generate a list of `load()`/`invalidate()` pairs to identical addresses to insure writebacks of old data occur when necessary. The `load()` may evict the previous occupant of the way, which is written back if necessary, and the `invalidate()` clears the block so it may be brought in by a `load()` or `store()` call. When less than *ways* elements are present, ordinary loads and stores are issued, letting the cache's replacement policy dictate placement and retaining dirty lines, if present.

When reconstructing lines that should be marked dirty, we call the `store()` method in timestamp order, oldest to newest. Every `store()` is followed by an `invalidate()` to all other caches. Clean lines are brought into the cache using calls to `load()` in timestamp order. Lines that `FixupCaches` has marked as invalid generate a series of `invalidate()` calls to the detailed cache.

By "replaying" the relevant accesses, the reconstruction process defers to the cache for all replacement decisions rather than exposing such policies to the reconstruction algorithm or requiring fine-grained read/write access to the cache model. For example, the CSR could store more than w ways and invoke a longer stream of loads and stores. By including additional memory references, one could prime a pseudo-LRU tree [103].

While reconstruction via replay is similar to the FFW approach, the MTR has a few advantages. We can store FFW-gathered state in a checkpoint, but we can only reconstruct caches that reflect a predetermined structure and predetermined replacement and coherence policies. MTR allows other replacement policies and a better estimate of coarse grained directory protocols because microarchitectural state is rebuilt using relative time stamps rather than with cache metadata. When used for online sampling, the MTR only runs functional cache/directory simulator on a small subset of requests while FFW runs it on every memory access.

3.5 Related Work

The concept of microarchitecture-independence is reflected in stack algorithms which allow single-pass simulation of multiple cache associativities [70]. Each set, $1..k$ is represented by a list called an LRU Stack. Addresses are added to the head of the list. When an address is accessed a second time, its depth, n, in the stack (or position in the list) indicates that the access is a hit in all caches with size $\leq n$. The address is removed and placed on the top of the stack; entries above the vacancy are pushed down to fill it. While this update procedure is not directly supported by a canonical stack, the elements make up an "LRU Stack" with the most recently accessed element at the top. The term "stack simulation" refers to the collection of elements, rather than the operations supported by the structure. The inclusion property

is initially demonstrated for fully associative caches of varying size ($k = 1$), then extended to set-associative caches. While various associativities are supported in a single pass, each cache must possess an equal number of sets due to an assumption of identical set mapping functions.

To simulate M different associativities ($m_1 \leq m_i \leq m_M$) at the same time requires M logical collections of stacks, each collection containing a stack for each set in the m_i-way associative cache (though tag storage may be shared among the collections) [46]. To further extend the stack algorithm, a set-refinement property was described that supported arbitrary *refining* set-mapping functions rather than merely a mapping based on least-significant-bits [46]. This allows single-pass simulation of direct-mapped and a larger range of set-associative caches. The extended algorithms, referred to as "forest simulation," support direct-mapped caches, while a generalized version, "all-associativity simulation," models the spectrum from direct-mapped to fully associative. It is embodied in a tool called Tycho. All-associativity simulation has a longer theoretical runtime than stack algorithms, but runs at similar speeds in practice. Later, a binomial tree algorithm reduced theoretical runtime from exponential to linear and was found to run faster on practical traces [110]. The binomial tree algorithm is used in a tool named Cheetah.

Stack analysis has been extended by Thompson to calculate the number of writes avoided by write-back caches in addition to standard miss rate metrics [114]. Whether a block has been written back depends on the size of the cache that contains/contained it. However, Thompson proved an inclusion property for dirtiness: if "a block is dirty in a cache of size C, then it is dirty in ... all larger caches." A *dirty level* is stored for each block that indicates the smallest cache size for which the block is dirty. When a block to be written is dirty at some level, a write is avoided in all caches greater than or equal to that level. When the dirty level is less than the valid level, it implies that the block is dirty in some cache sizes, but invalid (written back from) caches that are too small to contain it. When the dirty level is greater than the valid level, the block is clean in the relatively small caches with sizes between the dirty and valid level because the dirty block has been evicted and brought back in the clean state. A cache that is larger than the dirty level does not need to evict the line and retains it in the dirty state.

Thompson continued by extending stack analysis to support single-pass simulation of multiple configurations for multiprocessors. In the sense that he uses a single data structure to manage cache coherence state in a microarchitecture-independent fashion, his work has similar goals to ours, but it was proposed in a different context (online determination of memory system metrics with no sampling), and it uses a difference approach (stack analysis rather than timestamping and reconstruction). He observed that a block's coherence state in MOESI-based protocols (e.g., whether it is modified, owned, exclusive, shared, or invalid) can be determined by combining three properties: validity, ownership, and exclusivity. Ownership is based on the dirtiness of a block: a cache that modifies a block is said to own that

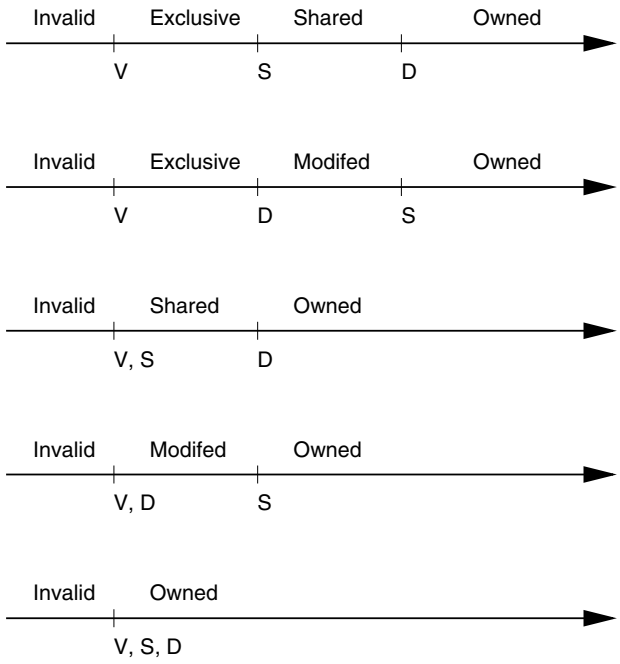


Figure 3-21. A figure from Efficient Analysis of Caching Systems showing how a block's state is determined based its levels and the size of its cache [114].

block. The dirty level used in the write-back analysis can be used to determine whether a block is owned in a cache. Like our MTR, the multiprocessor stack-based scheme takes advantage of the coherence property that there can be at most one outstanding dirty copy of any block.

Thompson proved that the valid and dirty inclusion properties hold in a collection of useful multiprocessor settings. An additional property regarding sharing inclusion was proven and used to determine whether a block is shared or exclusive. To take advantage of these theorems, a dirty level, an owner, and a sharing level is maintained for each block along with the valid level that is implicit in the stack representation of a set. A *level* indicates the smallest cache for which a property holds. As new blocks are pushed onto the stack, the valid level of existing blocks increases. The levels are updated on every memory access according to the parameters of the cache coherence protocol. For a given cache size, the relationship between the levels determines the state of a block. Figure 3-21 duplicates a figure from Thompson's thesis to illustrate the determination of block state given a cache size and block levels. The figure shows five legal level configurations. The levels for each block are shown as points on an abstract number line (e.g., V to the left of S indicates that the valid level of the block is less than its sharing level). Above the line are listed the possible coherence states of the block. Given a cache size, C, one makes the determination of a block's coherence state by choosing the line that represents the relationship of the block's levels, placing C on the number line, and consulting the coherence state at C. For example, if $V < S < D$ (the first configuration in the figure) and $V \leq C < S$, the block is clean and held Exclusive in this cache.

While multiprocessor stack simulation achieves many of the MTR's goals, there are several complications. It would be difficult to invoke in a parallel simulation as each cache must analyze each memory access as it is issued. Since no record of the access persists after a cache update, it would be difficult to reconstruct a directory system that allowed silent drops. Silent drops can reduce coherence traffic by allowing entries in a sharing vector that are no longer present in the cache. A multiprocessor stack analysis may be restricted to same-size caches for certain coherence protocols. While rare today, the trend toward heterogeneous processors could increase the significance of this restriction. The MTR does not have this problem as it can be coalesced in many ways. In additional to these particular advantages, the MTR has additional benefits in terms of size and snapshot generation speed.

The MTR is used as a self-compressing snapshot whereas the stack-based algorithms are used for online performance analysis. Though they share the goal of microarchitecture-independence, the usage differs. It would be possible to run a stack algorithm during functional simulation and use the contents of its structures as a MINSnap, but this would require fixing a maximum cache size and appears to require more storage than an MTR. We estimate the size of each structure using the calculations shown in Table 3.12. A stack simulator for a fully associative cache requires a tag for each element in the stack and a stack for each CPU in the target. Set-associative caches with less than w ways are represented by a set of pointers to the stack representing the fully associative cache [46]. For every block in the largest representable cache, maintaining coherence requires storing sharing level, dirty level, and the identity of the CPU that contains the dirty block. The size of the MTR is described in Section 3.1.1. Recall that the MTR contains an entry for every block referenced between detailed samples while a stack simulator's size is bounded by the largest cache it can represent. With the number of blocks touched per sample likely to be less than the maximum supported cache size, the size of the stack-based structures would be also be proportional to the number of touched blocks (rather than cache size), and the MTR would have an uncompressed size about 10% smaller than the size of multiple stacks. The MTR has a size advantage because the state of a line need not be stored as it is not determined until reconstruction. Likewise, no pointers are needed to support caches of varying associativity because the coalesce phase can determine each set's occupants during reconstruction. On the other hand, for targets and benchmarks for which the MTR leads to many lines in an ambiguous state, it may be worth sacrificing storage space and use the algorithms suggested by Thompson to eliminate the ambiguities.

Delaying cache reconstruction provides the MTR with a fast snapshot creation phase as it consists of simple, table updates with locality that corresponds to the target application. With stack algorithms, each memory reference requires more complex stack analysis, though this cost is amortized when snapshots are used.

Combining microarchitecture-independent structures with sampling of traces was suggested by Conte [25] and has recently reappeared in the context of execution-driven sim-

Storage required for stack-based one-pass analysis of multiprocessor caches			
tags	all-associativity pointers	state per block (dirty cache, sharing level, dirty level)	total
$4nc$	$4(w-1)$	$c(4+4+1)$	$4nc+9c+4w-4$
Storage required for MTR			
read timestamps	writer	write timestamp	total
$4nt$	t	$4t$	$4nt+5t$

Table 3.12. Amount of state in stack-based and MTR snapshots. Assuming four byte tags, pointers, level counters, and timestamps; one byte to indicate CPU (supports up to 255 processors). c=cache size, n=number of CPUs, w=ways (associativity), t=touched blocks.

ulation. The lengthy warming phase which precedes a detailed, execution-driven sample point may be amortized across many experiments by operating from stored snapshots which support various uniprocessor microarchitectures [125, 119]. Stack algorithms or structures similar to the Cache Set Record are used to represent state of all caches below a maximum size. Prior work suggests that a snapshot contain the state of every branch predictor configuration that might eventually be examined, but this can become prohibitive as we discuss in Chapter 4. Instead, we propose using specialized branch trace compression.

Solutions to the cold-start problem discussed in Section 2.3.3 must be adapted when sampling is used with single-pass algorithms [25]. A "fill-flush" technique flushes the cache prior to each detailed sample. Then, fill references (those that are not already in the cache) are used to update the cache simulator state, but they are omitted from the statistics calculation. This method ignores some memory references rather than risk determining their impact inaccurately. When the entire trace is available, references that do not occur during a detailed sample can be used to avoid any state loss at the expense of increased time. Statistics independent of cache associativity (number of references and recurrences) are computed using the entire trace, while misses that depend on size and associativity are accounted for during the sample with a single-pass simulation algorithm. This "no state loss" approach results in less error than fill-flush for smaller caches; it eliminates error altogether for larger caches.

Whole Execution Traces (WETs) are a compact, lossless representation of program behavior formed by annotating a program with addresses and data and dependency information (control and data) gathered at runtime [131]. The WET is represented as a graph with nodes for each basic block. Its edges contain control flow and dependency profiles for each statement in the block. Timestamps are used to identify dynamic instances of each basic block. In practice, the number of timestamps can be reduced by associating the same time to a group of basic blocks along the same acyclic path. Though the WET format is useful due to its compact size and its capability for bidirectional traversal, it does not account for programs run on a multiprocessor. Though one could extract memory access information from the WET, warming a single cache in the same way we use branch traces to warm a branch predictor, a WET does not attempt to represent microarchitectural structures.

Traces of committed instructions are inherently microarchitecture independent and, when used carefully, can sometimes replace the need for execution-driven simulation. The PHARMsim project desired to use deterministic traces so that one full-system multiprocessor simulation could be compared to another [64]. Their traces consist of memory accesses annotated with logical timestamps and dependency information to preserve the interleaving of memory accesses from different processors. External interrupts are recorded in the trace along with a logical timestamp. Time is measured using a count of committed instructions per CPU so that the interrupt is aligned with work rather than with cycles. If a microarchitectural change increases performance in spite of the delay injected to preserve memory dependencies, we know the change is beneficial. If the change hurts performance, the result is inconclusive unless the degradation is greater than the injected delay. The *determinism delay* metric is useful for judging whether the collected trace reflects the target machine. When many cycles have to be injected in order to preserve determinism, it may indicate that the stored trace is not representative of the current model.

Sun Microsystems recently presented an alternative approach: the Rapid, Accurate Simulation Environment (RASE) [30]. RASE uses a carefully validated simulator of an existing M-way platform to produce traces which are then duplicated to drive an N-way ($N > M$) chip-multithreaded simulation target. The simulation is fast because it is trace-driven rather than execution-driven. It is accurate because the traces are rigorously validated using real systems. RASE works well with applications, such as commercial database workloads, that scale-up linearly when run with additional processors and have a single phase of execution. Duplication involves shifting the non-shared address space to a non-overlapping region of memory, while shared segments remain shared. Such artificial replication could lead to timing variability due to new thread interactions, but no such variability is observed in practice. Random sampling need not be used when evaluating an application with a single phase of execution, but tens of millions of instructions are required to overcome cold-start effects and detect the result of architectural changes.

3.6 Summary

We have introduced the Memory Timestamp Record (MTR) and algorithms for its use. Despite its extensive description, the MTR is a simple concept which exploits fundamental properties of caches and cache coherence. Online sampling with the MTR enables faster execution of multiprocessor simulations: up to 1.45× faster than a multiprocessor functional warming model (FFW) and 7.7× faster than our detailed baseline. In addition, the MTR snapshot representation is not tied to specific microarchitectural details. We have shown how it can be used to reconstruct multiple cache configurations and coherence protocols. An MTR-enabled simulator will allow computer architects to evaluate a wide range of complex multiprocessor architectures with one snapshot per sample, rather than one snapshot per sample per target.

Chapter 4

Branch Predictor-based Compression

The previous chapter discussed how a single structure could represent the state of many caches and directories in a multiprocessor. Here we address the issue of representing a second type of structure which requires warming for accurate simulation: the branch predictor tables used by modern pipelined processors to mitigate the effects of control hazards. The main contribution of this chapter is a Branch Predictor-based Compression (BPC) scheme, which exploits a software branch predictor in the compressor and decompressor to reduce the size of the compressed branch trace snapshot. We show that when BPC is used, the snapshot library can require less space than one which stores just a single concrete predictor configuration, *and* it allows us to simulate *any* sort of branch predictor.[1]

A taken branch diverts a program from its current path, while a not-taken branch lets a program "fall-through" to the next instruction. The decision to take a branch does not occur until the branch has progressed several stages into a processor pipeline. In the meantime, to keep the pipeline full and the computer working at peak efficiency, the computer must continue to fetch instructions and begin their execution. If a branch deep in the pipeline is resolved to be taken, those speculatively fetched instructions must be discarded and their results nullified. An alternative is to stall execution until a branch is resolved. Both choices result in an underutilization of machine resources; had the correct instructions been fetched, no squashing or stalling would be necessary.

Accurate branch predictors reduce the number of cycles lost to wrong-path execution and are a popular area of study. Out-of-order superscalar processors are continually evolving their prediction strategies and simpler cores can now choose among many well-understood branch predictor options. To speed the evaluation of new and existing branch predictor structures, we would like to use microarchitecture-independent snapshots.

[1]The material in this chapter is based on the joint work of Kenneth Barr and Krste Asanović. The work originally appeared in the International Symposium on Performance Analysis of Systems and Software held in March of 2006 [7].

In a multiprocessor system, the issues of snapshot size and speed-of-use are multiplied by the number of processors, N, being simulated. Thus, it is crucial to have a technique that is as small and as fast as possible for flexible warming of a single branch predictor. Every byte of storage saved for a uniprocessor snapshot is N bytes in a multiprocessor snapshot; every minute of warming eliminated is N minutes that can be avoided in a multiprocessor simulation.

We begin with a review of branch predictor organization. Section 4.2 then explains why an MTR-like structure cannot be used to summarize the state of a predictor and why we resort to what is effectively trace-based simulation. We describe the structure of our BPC compressor in Section 4.3. In Section 4.4, we examine the performance of our technique versus general-purpose compressors and snapshots in terms of storage and speed.

4.1 Branch predictor overview

Two pieces of information can be predicted on every branch: the target address and whether or not the branch is taken. The earlier a correct prediction is made, the fewer cycles are lost due to stalls or wrong-path computation.

Typically the target address is looked up early in the pipeline by consulting a Branch Target Buffer (BTB). As with a cache, indexing is performed with low-order bits, and high-order bits are used to perform a tag match. If an instruction address is found in the BTB, the target from the BTB is used to redirect the fetch stage. The BTB is updated at branch target resolution and only stores targets of taken branches to save space. A tagged structure is used so that we do not attempt to redirect fetch on non-branch instructions. Figure 4-1 shows a direct mapped BTB, although most BTBs use associativity to reduce conflict misses. To improve prediction accuracy for indirect branches, in which the target is computed rather than encoded in the instruction, BTBs can be augmented with various schemes including loop predictors [97] and path-based predictors [31].

Later in the pipeline, when the instruction has been decoded and is known to be a branch, we can consult the untagged direction predictor. When predictive structures are accessed in several pipeline stages, there can be several misprediction penalties. For example, the Alpha 21264 stored line and way predictions in the L1 cache [53]. These predictions could be overridden by a more complex, more accurate branch predictor. The single-cycle penalty resulting from a correctly overridden prediction is small compared to the full seven-cycle branch misprediction penalty.

Figure 4-2 shows a canonical two-level adaptive direction predictor [116]. This predictor contains a branch history table (BHT) and one or more pattern history tables (PHTs). The BHT contains one or more branch history registers (BHRs) that store the outcomes (taken/not-taken) of prior branches. The BHR is a shift register: to update it, all bits are shifted, dropping the oldest outcome. A bit denoting the most recent branch outcome is

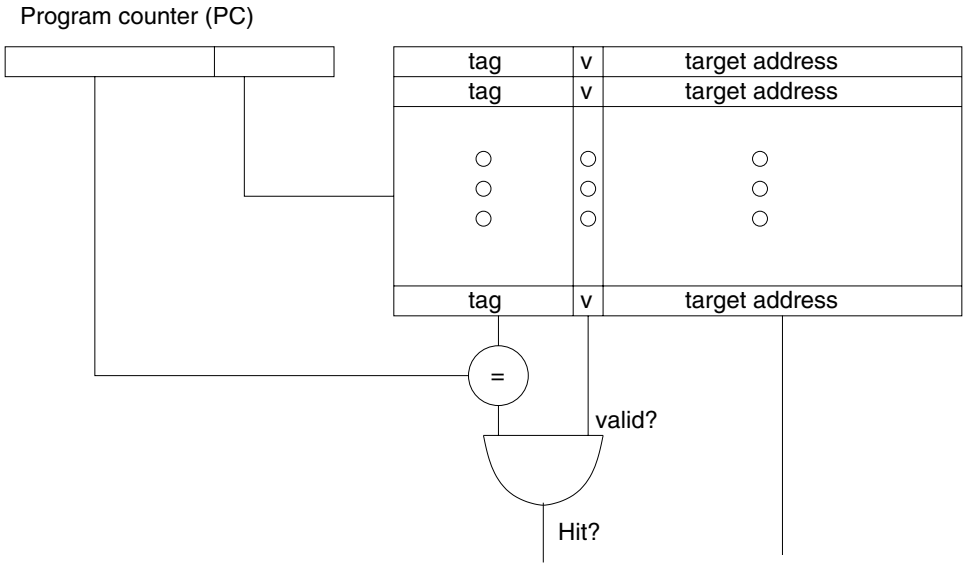


Figure 4-1. A branch target buffer (BTB). A canonical BTB is updated only with taken branches. Thus, hits in the BTB indicate the lookup PC is a taken branch.

shifted in. The program counter (PC) is used to choose one or more BHRs. This BHR is combined (optionally) with the PC to choose a prediction from a PHT. A PHT is a group of direction predictions. Usually, predictions are represented by the most significant bit of a saturating counter (e.g., 1=taken, 0=not-taken). Counters are usually at least two bits wide to provide hysteresis. For example, a two-bit counter becomes saturated "strongly taken" during a for-loop and will mispredict the final not-taken branch, but this misprediction brings the counter into a "weakly taken" state, preventing it from mispredicting the next occurrence of the taken loop branch. If there is more than one PHT, the PC is used to select a particular PHT.

4.2 Why can't we use a branch timestamp record?

While the BTB looks like a cache and can use MTR or stack-algorithm techniques for a microarchitecture-independent representation, indirect branch predictors and direction predictors are not as simple. The use of anti-aliasing hardware designs, the value of preserving historic branches, and the desire to keep snapshots small and microarchitecture-independent make stack algorithms infeasible and a "branch timestamp record" unappealing.

4.2.1 Anti-aliasing efforts complicate coalescing

To insure fast access times, branch direction predictors have limited capacity, which creates the potential for aliasing. Aliasing occurs in a branch direction predictor when multiple branches map to the same counter in the PHT. The aliasing can be destructive if a biased-taken branch is aliased with a branch that is biased not-taken.

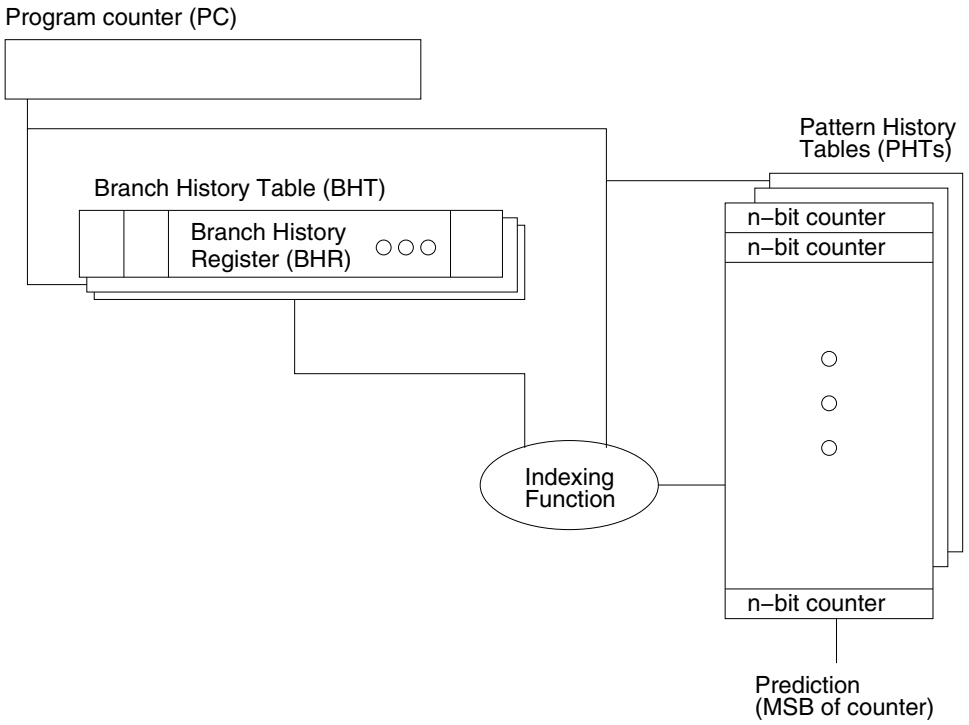


Figure 4-2. Parts of a canonical two-level adaptive branch predictor.

A second form of aliasing involves dynamic branches with the same static address, but different branch outcomes. A branch at the end of a loop is one example. It may be taken 10 times, but not-taken on the 11th occurrence. In this case, the outcome of a particular instance of the branch can be determined based on the context in which it appears. In the case of the loop-closing branch, we would predict taken unless the prior 10 outcomes were taken.

One way to reduce this form of aliasing is by combining the branch address with global history. This is shown in Figure 4-3 in which a single branch (at address 0x2400) is mapped to two different PHT entries depending on context. The figure depicts the use of an XOR operator to hash the branch address with the global history; other hash functions are acceptable. The hash function also results in a more efficient use of space, spreading branches across the entire table.

Figure 4-3 helps illustrate why MTR-like structures cannot directly be applied to branch direction predictors that incorporate history. Simple coalescing of addresses is insufficient to find all branches that map to a particular counter because the hash with history bits may have forced additional branches to the same counter. Also, dynamic instances of the same branch are often mapped to different counters, so an address-based organization is inappropriate.

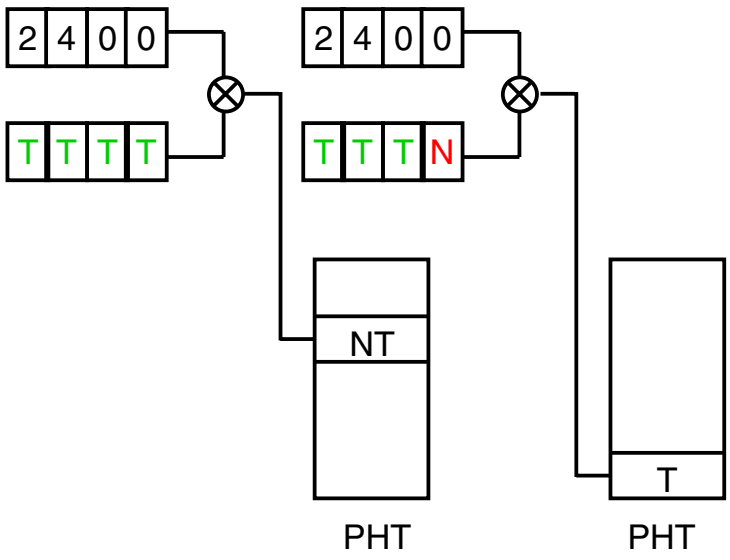


Figure 4-3. Anti-aliasing maps a single branch to multiple PHT entries depending on context.

initial state	state after outcome T NT T ... NT T				possible predictions
0	1	0	1	...	NT
1	2	1	2	...	NT, T
2	3	2	3	...	T
3	3	2	3	...	T

Table 4.1. Role of initial counter state in predicting direction.

4.2.2 Long-lived history is helpful

In the MTR, we can save space by overwriting least-recently-used entries, as these are the ones that would be evicted by a cache using time-based replacement. In a branch predictor, the use of history means that the address and outcome of old branches can play an important role in determining the prediction of new branches. Maintaining individual counter history is important as well. If a single counter in the PHT undergoes a long period of alternating taken/not-taken, it is crucial that we have stored enough previous branches mapping to this counter in order to determine, from its most recent confirmable state, whether it now predicts taken or not-taken. This phenomenon, where a final prediction depends on initial state, is shown in Table 4.1.

4.2.3 Certain structures are difficult to generalize

One possible way to create a MINSnap for two-level predictors with one BHR and one PHT is to identify each branch by its address and the global history at the time it was encountered. This is not truly microarchitecture independent as it fixes an upper bound on the size of

the global history register. Smaller histories are supported by considering, at reconstruction time, only the most recent history. Each < branch,history > pair could be used as a key to index into a hash table. The key's value is a time-ordered list of booleans indicating taken or not-taken. To reconstruct a two-level adaptive predictor with a global history register, we determine the size of the BHR and PHT and the indexing function (a function of the PC and history). For every populated entry in the hash table, we perform the indexing function on the < branch,history > key, coalescing entries that have the same PHT index. We consult the coalesced direction list in reverse order until we are certain of the counter's state (e.g., if the three most recent branches are taken, a two-bit counter must be saturated). If we reach the oldest branch without discovering a saturating sequence (e.g., an alternating pattern of taken/not-taken), then the predicted direction of this branch is ambiguous without knowledge of the counter's initial state. The lists could be gradually pruned if the size of the saturating counter was known in advance. While this scheme is functionally correct, it requires a lot of state and considerable time to perform reconstruction. Most predictors are not so simple to represent.

Some branch predictors, such as the "local" component of the Alpha 21264, use multiple BHRs in which certain branches share a specific history register. In such predictors, we would have to coalesce sharers of a history register prior to determining to which counter a branch is mapped at a particular point in time. Not only does this drastically increase the amount of state in the timestamp record, but the multiple coalescing is now more complicated than the branch predictor itself. Working with tournament predictors, which include a "chooser" to select between two predictors, is even more difficult as the reconstruction process must somehow determine the value of the chooser counter by determining the predictions of each component predictor at the time prior to updating the chooser.

The newest branch predictor proposals break away from the use of a PHT with simple saturating counters, relying on structures called perceptrons to learn a branch's behavior and predict its outcome [51]. A hash of the branch address is used to select a perceptron from an array. Each perceptron contains a collection of positive and negative weights. The dot-product of these weights and the bits of the global history register is computed to produce a prediction. A "0" in the global history is treated as "-1" when performing the dot-product; a positive dot product indicates taken, and negative dot products are not-taken. A training threshold (which is dependent on the width of the weights) dictates whether or not the perceptron is updated as a result of a branch. Since the state of a concrete predictor is comprised of its weights, and the weights can increment or decrement with every branch in the program (subject to training threshold), it appears that an entire branch trace is needed to reconstruct any given perceptron predictor. Imposing a maximum branch history length, perceptron count, and weight width might allow coalescing, but makes the structure decidedly microarchitecture-dependent. Such a reconstruction algorithm might process as follows. Once the hash function used to select predictors is determined, perceptrons mapped at the

Parameter	Size	Example
predictor size	8 KB	gshare predictor with 15 bits of global history and 2-bit counters
samples	1000	1B instructions of application sampled every 1M instructions
other predictors	10	
benchmarks	26	SPECCPU 2000
processors	16	
Total	32 GB	

Table 4.2. High storage costs for branch predictors in sampling simulation.

same hash table index are coalesced by summing corresponding weights and truncating the values according to the width of each weight. When there are more weights than there are history bits of the concrete predictor, the excess weights are discarded. This algorithm does not support a threshold parameter as it is unknown which weights will be combined until the indexing function is determined (i.e., we cannot know when the threshold is exceeded); also, the ideal threshold depends on the history length which is not known until reconstruction time.

4.3 Design of a branch predictor-based trace compressor

Previous work suggested that snapshots should contain the microarchitectural state of every predictor that might be examined [119, 125]. Unfortunately, this limits flexibility and increases snapshot size, particularly when many samples are taken of a long-running multiprocessor application. A back-of-the-envelope calculation shows that this can quickly become unreasonable. The conservative parameters in Table 4.2 lead to 32 gigabytes of required storage — and this is before the inclusion of branch target predictors.

To evaluate a predictor that has not been stored, one must regenerate the snapshot library, forfeiting the time-savings expected of pre-generated snapshots, or suffer the effects of an inaccurately-warmed predictor. We advocate an alternative approach in this chapter, which is to store a compressed version of the complete branch trace in the snapshot. This approach is microarchitecture-independent because any branch predictor can be initialized before detailed simulation begins by uncompressing and replaying the branch trace.

In general, lossless data compression can be partitioned into two phases: modeling and coding [94]. The modeling phase attempts to predict the input data symbols. For each symbol in the input text, the compressor expresses any differences from the model. The coding phase creates concise codewords to represent these differences in as few bits as possible. BPC uses a collection of *internal predictors* to create an accurate, adaptive model of branch behavior. We delegate the coding step to a general-purpose compressor.

To model the direction and targets of branches in a trace, we can draw on years of research in high accuracy hardware branch predictors. When using branch predictors as models in software, we have two obvious advantages over hardware predictors. First, the severe

constraints that usually apply to branch prediction table sizes disappear. Second, a fast functional simulator (which completes the execution of an entire instruction before proceeding) can provide oracle information to the predictor such as computed branch targets and directions. We use the accurate predictions from software models to reduce the amount of information passed to the coder. When the model can predict many branches in a row, we do not have to include these branches in the compressor output; we only have to express the fact that the information is contained in the model.

4.3.1 Structure

Figure 4-4 shows the different components in our system. A benchmark is simulated with a fast functional simulator, and information about branches is passed to the BPC Compressor. The BPC Compressor processes the branch stream, filters it through a general-purpose compressor, and creates a compressed trace on disk. We will show momentarily how the BPC Compressor can improve its compression ratios by using its own output as input to compress the next branch.

We define a *concrete branch predictor* to be a predictor with certain fixed parameters. These parameters may include size of global history, number of branch target buffer (BTB) entries, indexing functions, etc. To recreate the state of various concrete branch predictors, we retrieve the compressed trace from disk, perform general-purpose decompression, and process the result with the BPC Decompressor. The structure of the decompressor is identical to that of the compressor. The branch trace output from the BPC Decompressor is replayed into one or more concrete predictors. Branches later in the trace will overwrite entries in the concrete predictor according to its policies.

The particular collection of internal predictors has nothing to do with the concrete branch predictors that BPC will warm. The implementation of BPC merely uses predictors to aid compression of the complete branch trace which, by its nature, can be used to fill *any* branch predictor with state based on information in the trace. Furthermore, the precise construction of a BPC scheme is up to the implementer who may choose to sacrifice compression for speed and simplicity or vice versa. We merely describe what appears to be a happy medium.

This implementation differs from other proposed value predictor-based compression (VPC) techniques, which feed several predictors in parallel and emit a code indicating which predictor is correct [15, 108]. For our datasets of branch traces, we have found that such a stream of predictor selection codes does not compress as well as the BPC trace with its single, statically determined predictor.

4.3.2 Branch notation and representation

Before explaining the details of the compressor, we describe the information stored in the compressed trace and introduce some notation used in this section.

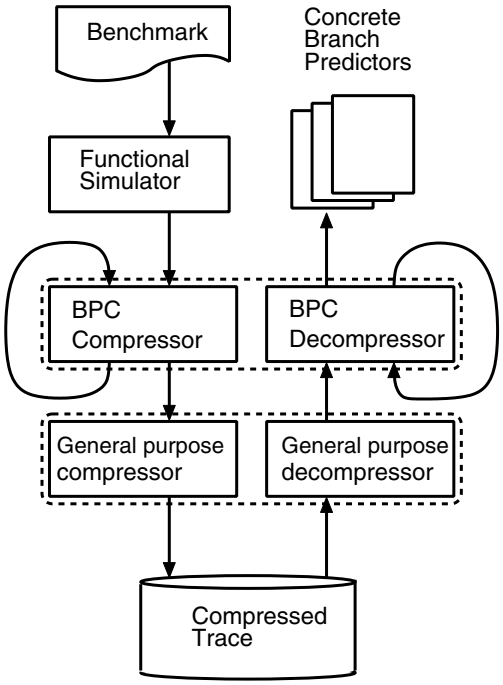


Figure 4-4. System diagram. The compressed trace is stored on disk until it is needed to reconstruct concrete branch predictors.

Using the Championship Branch Prediction (CBP) framework [108], the uncompressed branch traces in our study consist of fixed-length *branch records* as shown in Table 4.3. Each branch record contains the current instruction address, the fall-through instruction address (not obvious in CISC ISAs), the branch target, and type information that indicates whether the branch is a call, return, indirect, or conditional (not mutually exclusive). The branch records are generated by a functional simulator which can resolve the branch target and provide a taken/not-taken bit. The taken bit is stored in a one-byte field to facilitate compression via popular bytewise algorithms such as gzip [40], bzip2 [95], or PPMd [99].

field				size (Bytes)
instruction address				4
fall-through instruction address				4
branch target address				4
taken				1
is_indirect	is_conditional	is_call	is_return	1

Table 4.3. Format of branch records.

Rather than predicting the direction and target of the current branch, B_n, as in a hardware direction predictor and BTB, we predict information about the *next* branch, B_{n+1}. We denote the actual branch stream as $B_{1..k}$ and denote predicted branches as $\beta_{1..k}$. If the pre-

dictor proves correct (i.e., $\beta_{n+1} == B_{n+1}$), we concisely note that fact and need not provide the entire B_{n+1} branch record. Furthermore, we use β_{n+1} to produce $\beta_{n+2..n+i}$ for as large an i as possible. This allows us to use a single symbol to inform the decompressor that i chained predictions will be correct.

Figure 4-5 depicts an example using this notation. Given B_n, we must provide information about β_{n+1}. In this case, we have predicted β_{n+1} to be not-taken with a specific fall-through address. If these are correct predictions, BPC can continue by chaining: using β_{n+1} as input to request predictions about β_{n+2}. The longer the chain of correct predictions, the less information has to be written by the compressor.

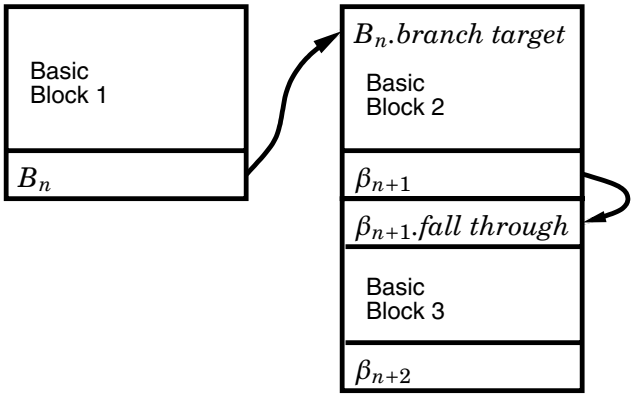


Figure 4-5. An example to illustrate our notation.

The output of the compressor is a list of pairs. The first element indicates the number of correct predictions that can be made beginning with a given branch; the second element contains the data for the branch that *cannot* be predicted. An example output is shown in Table 4.4. As in most branch traces, the example shows that the first few branches cannot be predicted and must be transmitted in their entirety. Eventually the compressor outputs B_{10} and uses B_{10} as an input to its internal predictors, coming up with a prediction, β_{11}. Comparing β_{11} to the next actual branch, B_{11}, a match is detected. This process continues until the internal predictors fail to guess the incoming branch (at B_{24}). Thus, we output "13" to indicate that, by chaining predictor output to the next prediction's input, 13 branches in a row will be predicted correctly, but $\beta_{24} \neq B_{24}$. We emit B_{24} and repeat the process.

We store the output in two separate binary files and use a general-purpose compressor (such as gzip or bzip2) to catch patterns that we have missed and to encode the reduced set of symbols. We also considered the less common compressor, PPMd. PPMd is a fast implementation of the PPM algorithm [23] that uses an arithmetic encoder and tends to produce better compression ratios than bzip2 in roughly equal time. While PPM-based methods are often discounted due to their slow speed, we found PPMd to perform faster than bzip2 for our source data. Of course, a mild speed penalty during the compression phase could be accepted as snapshot generation occurs just once. We used an order-14 model corresponding to the

skip amount	branch record
0	B_0
0	B_1
0	B_2
...	...
0	B_{10}
13	B_{24}

Table 4.4. Example compressor output.

number of bytes-per-record in the raw trace file; this corresponds to a 1st-order model at the branch record granularity. The model is reset when it reaches 32 MB.

4.3.3 Algorithm and implementation details

The internal structure of a BPC Compressor is shown in Figure 4-6. Each box corresponds to one predictor. When multiple predictors are present at a stage, only *one* is consulted. In BPC, the criteria for choosing a predictor stems from branch *type* which expresses characteristics of the branch such as whether it is a return instruction or whether it is conditional. The details of how type determines predictor selection are explained below.

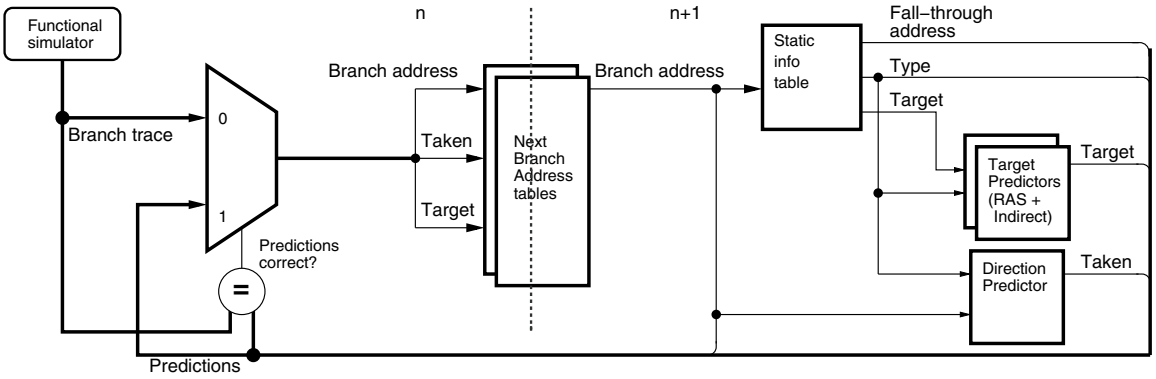


Figure 4-6. Prediction flow used during branch trace compression. Input left of the dashed line is from the current branch, B_n. To the right of the dashed line are predictions for the next branch, β_{n+1}.

The description below refers to the steady-state operation. We do not describe handling of the first and last branch, nor do we detail the resetting of the *skip* counter. These corner cases are addressed in our code. We use `diff` to validate that a compressed trace is uncompressed correctly.

Initially, a known address, target, and taken bit of B_n are received from the functional simulator and used to predict the address of β_{n+1}. This address is used to look up static information about β_{n+1} including its fall-through address, type, and target. In the absence of context switches or self-modifying code, the branch address corresponds directly with a type

and fall-through address. If the branch target is not computed, a given branch address always has the same branch target. The type prediction helps the direction predictor decide whether β_{n+1} is a taken branch. The type also helps the target predictor make accurate predictions. Once components of β_{n+1} have been predicted, it can be used to generate a prediction for β_{n+2} and so on. Before continuing, the predictors are updated with correct data for B_{n+1} provided by the simulator.

Predicting the next branch address. Normally, branch *targets* must be predicted, but in the case of BPC, they are known by the functional simulator. Instead, we must predict the address of the next *branch*. Since the next branch seen by a program is the first branch to appear after reaching a target, knowing the target allows us to know the next branch. If the branch is not-taken, the next branch should appear shortly a few instructions later. This prediction will be correct unless we are faced with self-modifying code or process swaps.

If B_n is a taken branch, we use a simple mapping of current branch target to next branch address. This map can be considered a Last-1-Value predictor or a cache. Our map is implemented with a 256K-entry hash table. The hash tables and fixed-size predictors of BPC provide $O(1)$ read and write time with respect to trace length. Since the hash table merely provides predictions, it need not detect or avoid collisions. This property permits a fast hash function and immediate insertion/retrieval, but we choose to use linear probing to avoid collisions and achieve higher prediction accuracy. When the hash table becomes full, we allow overwriting of the element currently stored at the hash key location.

If B_n is not-taken, we use a separate table indexed by current branch address to reduce aliasing. By using two tables we ensure that taken and not-taken instances of the same branch do not overwrite each other's next address mapping.

Recall that BPC is benefiting from the oracle information provided by the functional simulator. Hardware target predictors are accessed early in the cycle before the instruction has been decoded and resolved. Here, the simulator has produced $B_n.taken$ and $B_n.target$ which it uses to select and index the maps.

Predicting the next branch's static information. The compressor looks up β_{n+1}'s branch address in a hash table to find the type, the fall-through address, and a potential target for β_{n+1}. Note that this target may be overridden if the branch type indicates an indirect branch or return instruction. The lookup table is implemented as above.

Predicting the next branch's direction. If the predicted type indicates the next branch is conditional, we look up its direction in a direction predictor. Recall that our software predictors are not constrained by area or cycle time as in a hardware predictor. Thus, we chose a large variant of the Alpha 21264 tournament predictor [53]. Like the gshare predictor [72], we use the XOR operator to combine the program counter with a 16-bit global branch history. The result is an index used to access a global set of two-bit counters. We use 2^{16} local histories

(each 16 bits long) to access one of 2^{16} three-bit counters. A chooser with 2^{16} two-bit counters learns the best-performing predictor for each PC/History combination. This represents 1.44 Mbits of state, much more than one could expect in a real machine. Branches predicted to be non-conditional are marked taken.

Predicting the next branch's target. If the next branch is a return, we use a 512-deep return address stack to set its target. This extreme depth was necessary to prevent stack overflows in some of our traces that would have hidden easily-predicted return addresses.

If the next branch is a non-return indirect branch, we use a large filtered predictor to guess the target [32]. We introduce a 32 K-entry BTB leaky filter in front of a path-based indirect predictor. The path-based predictor is a 2^{20}-entry version of the predictor provided by the Championship Branch Prediction contest [108]. It has a PAg-like structure — each PC has its own path register which indexes a single address prediction table ([116]) — and uses the last four targets as part of the index function. The filter quickly learns targets of monomorphic branches, reducing the cold-start effect and leaving more room in the second-stage, path-based predictor.

If the next branch is neither a return nor an ordinary indirect branch, we set the target equal to the last target found in the hash table of static branch information.

Emitting the output stream and continuing. The β_{n+1} structure created thus far is compared with the actual next branch, B_{n+1}. If they match, we increment a *skip* counter; if not, we emit $< skip, B_{n+1} >$. To keep a fixed-length output record, we do not allow *skip* to grow past a threshold (e.g., a limit of 2^{16} allows the skip value to fit in two bytes).

Before repeating, all predictors are updated with the correct information: for each B_n, the instruction address tables are indexed by B_n's address or target address and updated with B_{n+1}'s instruction address, while the remaining predictors are indexed by B_{n+1}'s instruction address and updated with B_{n+1}'s resolved target, taken bit, fall-through address, and type. Finally, we increment n and repeat the above steps.

Entropy coding. The output of the BPC compressor is significantly smaller than the original branch trace, but better results are possible by employing a general-purpose compressor such as gzip, bzip2, or PPMd. These highly tuned compressors are sometimes able to capture patterns missed by BPC and use Huffman or arithmetic coding to compress the output further.

4.3.4 Decompression algorithm

The decompressor must read from the *skip amount* and *branch record* files and output the original branch trace one branch at a time and in the correct order. The BPC decompression process uses the same structures described in Section 4.3.3 so it can be described quickly. As above, we assume a steady state where we have already read B_n.

After reversing the general-purpose compression, the decompressor first reads from the *skip amount* file. If the skip amount is zero, it emits B_{n+1} as found in the *branch record* file, and updates its internal predictors using B_n and B_{n+1}.

If it encounters a non-zero skip amount, it uses previous branch information to produce the next branch. In other words, to emit B_{n+1}, it queries its internal predictors with B_n and outputs the address, fall-through, target, type, and taken information contained in the internal predictors. Next, the skip amount is decremented, B_{n+1} becomes the current branch (B_n), and the process repeats. Eventually, the skip amount reaches 0, and the next branch must be fetched from the input file rather than emitted by the predictors.

As the decompressor updates its internal predictors using the same rules as the compressor, the state matches at every branch, and the decompressor is guaranteed to produce correct predictions during the indicated skip intervals. The structure of the decompressor is identical to that of the compressor, so decompression proceeds in roughly the same time as compression.

4.3.5 Usage

Recall that the motivation for compressing a branch trace is to replace concrete branch predictor snapshots for sampling-based simulation. By piping the output of our decompressor into a concrete branch predictor model, the model becomes warmed-up with exactly the same state it would have contained had it been in use all along. Furthermore, the decompressed branch stream can be directed into *multiple* concrete branch predictors so that each may be evaluated during detailed simulation. After each branch has been observed by a concrete predictor, it is no longer required. Thus, no additional storage is needed on the simulation host to hold the uncompressed trace. Note that some detail is lost since the PHT is not updated with wrong-path branches, and a trace does not capture the effects of delayed update.

Speculatively updating branch predictors generally improves their accuracy as long as a mechanism exists to repair the effects of wrong-path updates [101]. Deep pipelines allow multiple branches to issue before earlier branches have been resolved, so speculative updating helps maintain any correlations between a current branch and its predecessors that have not yet been committed. While the high accuracy of branch predictors results in most of these updates applying to correctly predicted branches, experiments show that accuracy is badly degraded if the BHR contains updates from a wrong path. Therefore, the BHR is restored when a misprediction is detected, though the PHT typically retains wrong-path updates.

Delayed update refers to the fact that, in order to be effective, branch predictors are consulted several cycles before branches are resolved. The latency between lookup and update can cause a given branch to be mispredicted multiple times, instigating multiple pipeline flushes [66].

Despite the inability of traces to capture microarchitecture-dependent behaviors such as wrong-path updates and delayed updates, the proposed use of BPC-compressed traces is no

Name	Branches (Millions)	Insts/ Branch	Conditional	Return	Call	Indirect	Unconditional
			(percent of total)				
FP-1	2.6	11.3	84.6	5.5	5.5	0.0	4.4
FP-2	1.8	16.3	99.3	0.0	0.0	0.0	0.6
FP-3	1.6	18.7	98.3	0.4	0.4	0.0	0.9
FP-4	0.9	32.0	97.2	0.9	0.9	0.0	1.1
FP-5	2.7	10.8	89.0	4.6	4.6	0.0	1.8
INT-1	5.0	5.9	83.9	4.6	4.6	0.0	7.0
INT-2	3.7	8.0	78.1	6.2	6.2	0.8	8.7
INT-3	4.1	7.1	91.2	0.7	0.7	0.0	7.4
INT-4	2.4	12.1	85.1	5.8	5.8	0.0	3.3
INT-5	3.8	7.7	98.3	0.5	0.5	0.2	0.6
MM-1	2.8	10.6	80.1	5.2	5.2	0.0	9.6
MM-2	4.2	7.0	90.4	2.6	2.6	1.7	2.7
MM-3	5.0	6.0	60.9	16.7	16.7	0.1	5.7
MM-4	5.1	5.8	95.9	1.5	1.5	0.2	0.9
MM-5	3.4	8.7	75.3	8.9	8.9	2.6	4.3
SERV-1	5.6	5.3	65.3	12.3	12.4	0.4	9.6
SERV-2	5.4	5.4	65.0	12.3	12.3	0.4	10.0
SERV-3	5.4	5.5	71.1	8.3	8.3	0.2	12.0
SERV-4	6.3	4.7	67.7	10.3	10.3	0.3	11.3
SERV-5	6.4	4.6	66.9	10.4	10.4	0.3	12.0

Table 4.5. Characteristics of traces. Note that indirect branches refer to those branches not already classified as Calls or Returns. Unconditional branches are those that remain after classifying indirects, calls, and returns. Columns may not sum to 100% due to rounding.

worse than previously proposed schemes, and the wrong-path effect has been shown to be minor — usually less than 1% of CPI [127].

4.4 Evaluation

Our simulation framework is based on the Championship Branch Prediction (CBP) competition trace reader which provides static and dynamic information about each branch in its trace suite [108]. The trace suite consists of 20 traces from four categories: integer, floating point, server, and multimedia. Each trace contains approximately 30 million instructions comprising both user and system activity, and the traces exhibit a wide range of characteristics in terms of branch frequency and predictability as shown in Table 4.5 and Table 4.6. Columns labeled CBP show the direction accuracy and indirect target accuracy of the predictors used in the CBP trace reader: a gshare predictor with a 14-bit global history register, and an indirect target predictor in a PAg configuration with 2^{10} entries and a path-length of 4 (bits from the past four targets are hashed with the program counter to index into a target table). The BPC column shows the decreased misprediction rate available to BPC with the direction predictor and target predictor configuration described in Section 4.3.3.

Name	CBP	BPC	CBP	BPC
	Direction (Mispred. Rate)		Indirect Target (Mispred. Rate)	
FP-1	0.051	0.039	0.314	0.288
FP-2	0.018	0.017	0.317	0.303
FP-3	0.009	0.008	0.286	0.277
FP-4	0.010	0.010	0.251	0.241
FP-5	0.010	0.004	0.598	0.563
INT-1	0.053	0.049	0.362	0.337
INT-2	0.078	0.074	0.597	0.526
INT-3	0.106	0.094	0.313	0.285
INT-4	0.036	0.032	0.009	0.008
INT-5	0.005	0.003	0.285	0.250
MM-1	0.099	0.108	0.001	0.001
MM-2	0.079	0.079	0.015	0.011
MM-3	0.030	0.014	0.114	0.101
MM-4	0.011	0.011	0.053	0.046
MM-5	0.067	0.055	0.172	0.062
SERV-1	0.040	0.021	0.357	0.024
SERV-2	0.043	0.023	0.377	0.026
SERV-3	0.045	0.037	0.113	0.057
SERV-4	0.040	0.026	0.242	0.023
SERV-5	0.040	0.025	0.258	0.019

Table 4.6. Predictability of traces. For a description of the CBP and BPC predictors, please see text of Sections 4.4 and 4.3 respectively.

Using this framework we will show that BPC provides an excellent level of compression. Not only does a compressed trace require less space than compressed snapshots, but a BPC-compressed trace is smaller and faster to decompress than other compression techniques.

4.4.1 Compression ratio

Figure 4-7 shows the compression ratio resulting from various methods of branch trace compression for each trace. Traces were run to completion with snapshots taken every 1M instructions. This sampling interval was found to produce good results on SPECCPU benchmarks [127]. Each trace provides enough branches for 29 snapshots. We report bits-per-branch (rather than absolute file size or ratio of original size to new size) so that our results are independent from the representation of branch records in the input file. From left-to-right we see compression ratios for unprocessed branched traces (compressed with various general-purpose compressors); compressed concrete snapshots; VPC (a similar work which is discussed in Section 4.4.6); and BPC as described in this thesis. We use the suffix *+comp* to denote the general-purpose compressor used with each technique.

While slower, bzip2 and PPMd give astonishingly good results on raw trace files composed of fixed-length records. In fact, these general-purpose compressors use algorithms that have a more predictive nature than the dictionary-based gzip.

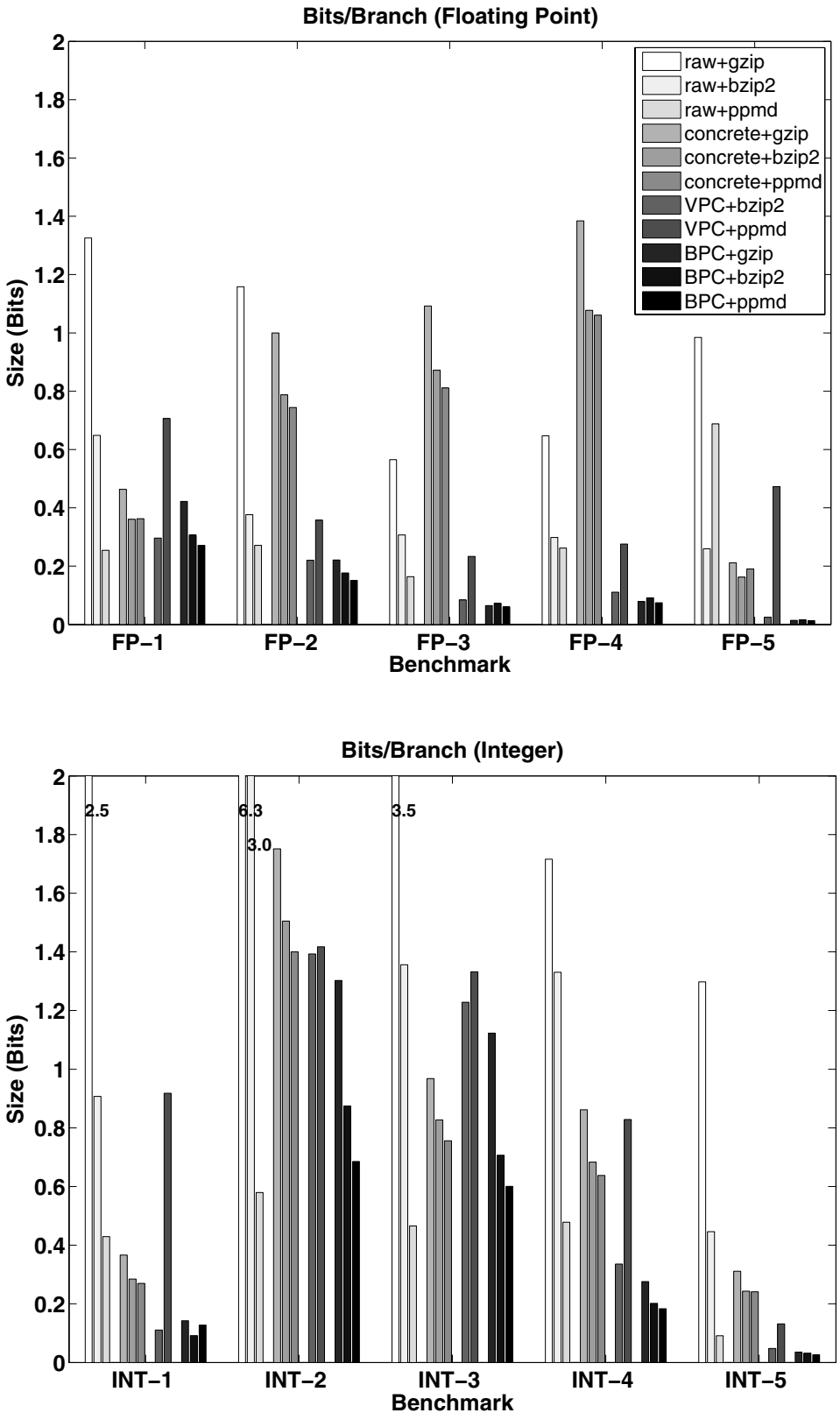
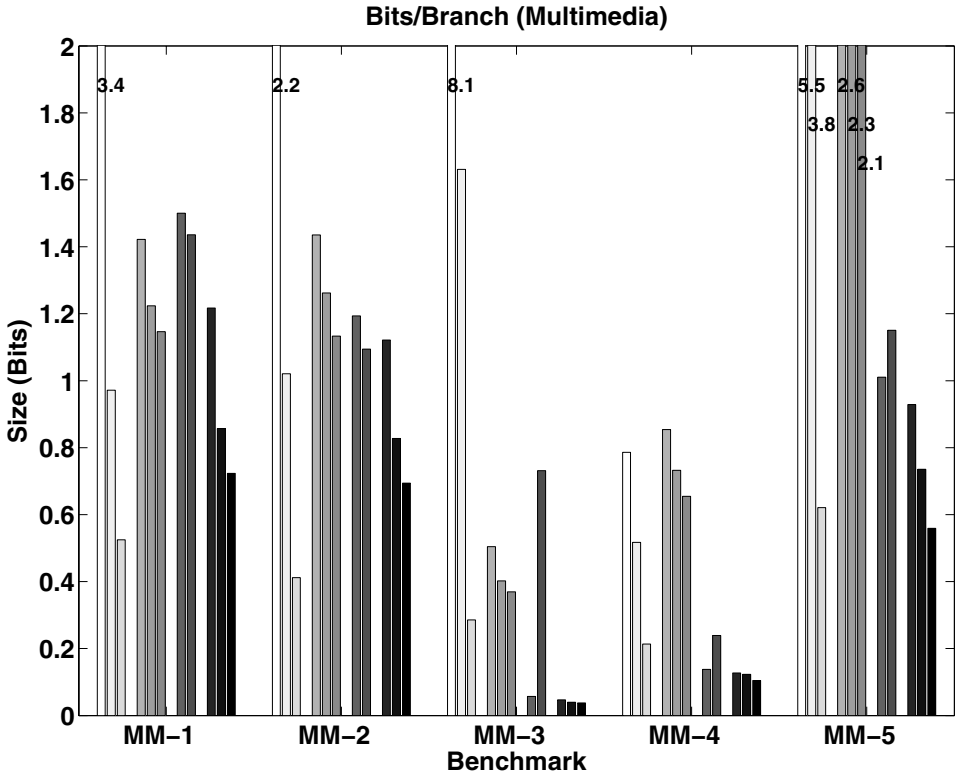


Figure 4-7. Compressed size (bits/branch). Large values clipped and indicated with number on bar.



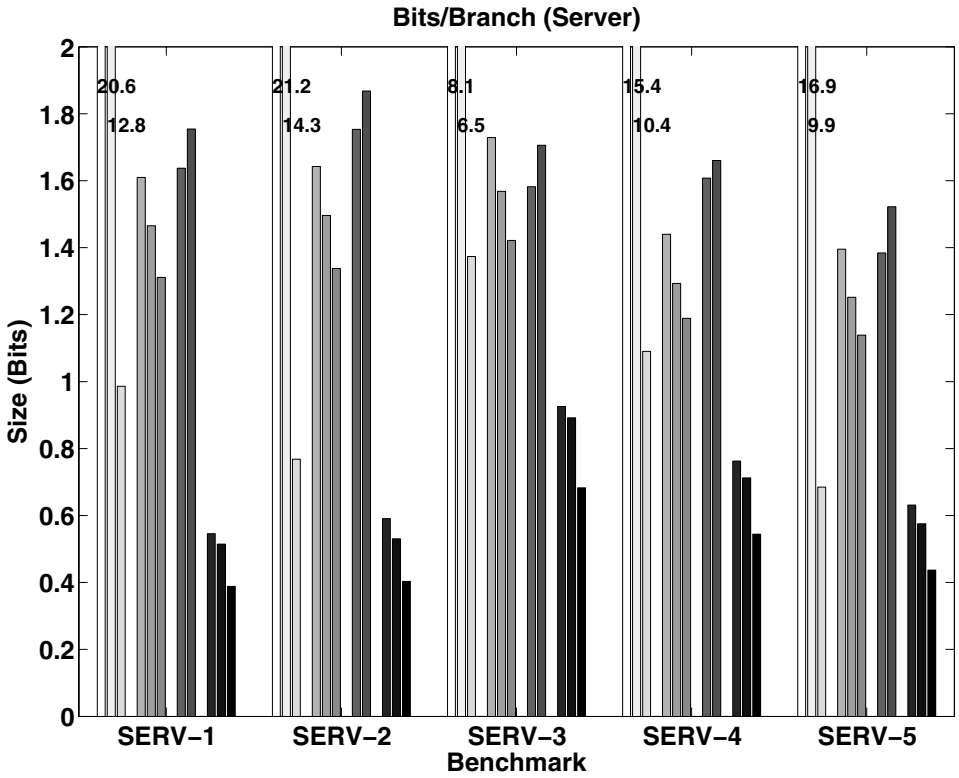


Figure 4-7 (continued)

	Space savings (%) Bytewise vs. Bitwise		
	gzip	bzip2	PPMd
FP-1	-0.8	13.0	1.4
FP-2	3.6	14.9	13.0
FP-3	15.2	20.9	22.8
FP-4	14.6	20.9	16.9
FP-5	10.6	20.6	-9.9
INT-1	2.5	14.9	12.5
INT-2	-3.9	10.4	10.8
INT-3	-7.9	5.9	4.7
INT-4	1.4	13.0	12.3
INT-5	8.7	17.2	8.7
MM-1	-14.1	3.3	-0.4
MM-2	4.4	13.0	16.9
MM-3	12.3	20.9	23.7
MM-4	10.1	19.1	23.3
MM-5	4.5	12.5	16.9
SERV-1	7.2	14.1	19.7
SERV-2	7.9	14.5	20.1
SERV-3	3.3	12.4	17.5
SERV-4	5.9	14.2	18.3
SERV-5	6.9	14.8	19.0

Table 4.7. Bitwise snapshots are smaller, but less compressible than bytewise snapshots.

The three bars labeled "concrete" show the size of a snapshot containing a single branch predictor roughly approximating that of the Pentium 4: a 4-way, 4096 entry BTB to predict targets and a 16-bit gshare predictor to predict directions [75]. Together the uncompressed size of the concrete predictor is 43.6 KB, however, we use a bytewise representation and store a 97 KB snapshot as it is more amenable to compression than a bitwise representation — up to 20% smaller in some cases. Figure 4-7 shows the size of bytewise snapshots after compression with gzip, bzip2, and PPMd. Table 4.7 shows the savings provided using bytewise snapshots instead of bitwise snapshots. With gzip compression, bitwise snapshots are larger in all cases except INT-2, INT-3, MM-1, and FP-1. When bzip2 is used, bytewise snapshots are always smaller. With PPMd, only MM-1 and FP-5 perform better as bitwise snapshots.

The state of a given branch predictor (a *concrete snapshot* in our terminology) has constant size of q bytes. However, to have m predictors warmed-up at each of n detailed sample points (multiple short samples are desired to capture whole-program behavior), one must store mn q-byte snapshots. Concrete snapshots are hard to compress so p, the size of q after compression, is roughly constant across snapshots. Since a snapshot is needed for every sample period, we consider the cumulative snapshot size: mnp. This cumulative snapshot grows with m and n. In fact, it grows *faster* than a BPC-compressed branch trace even for reasonable p and $m = 1$. Table 4.8 shows that combining the concrete snapshots before compression provides context which is helpful for compression. The first three columns compare the total

	Combine snapshots, then compress. Size savings (%) vs. compress+combine			Combine snapshots, then compress. Size savings (%) versus BPC		
	gzip	bzip2	PPMd	gzip	bzip2	PPMd
FP-1	0.3	77.7	23.5	-9.4	73.7	-2.3
FP-2	0.2	78.5	31.6	-351.2	4.0	-236.1
FP-3	0.4	89.3	41.7	-1584.4	-29.7	-689.8
FP-4	0.1	85.3	36.2	-1657.4	-73.4	-806.8
FP-5	-0.6	85.3	22.5	-1407.3	-47.1	-1018.7
INT-1	0.2	84.1	28.4	-156.2	50.7	-51.7
INT-2	-0.1	39.3	46.1	-34.6	-4.4	-10.2
INT-3	-0.0	54.1	38.5	13.8	46.4	22.5
INT-4	0.2	82.9	30.8	-212.1	42.1	-141.0
INT-5	0.5	86.3	29.7	-766.7	-4.7	-541.6
MM-1	-0.1	44.3	36.5	-16.9	20.5	-0.6
MM-2	0.0	55.8	55.2	-27.9	32.5	26.9
MM-3	0.2	83.5	43.9	-978.2	-67.0	-447.7
MM-4	-0.1	78.3	60.4	-574.3	-29.0	-147.8
MM-5	-0.1	42.9	56.3	-176.4	-81.4	-65.1
SERV-1	-0.0	44.3	56.4	-194.9	-58.4	-47.1
SERV-2	-0.0	43.7	53.5	-178.2	-58.7	-54.4
SERV-3	-0.1	31.7	45.1	-87.0	-20.0	-14.2
SERV-4	-0.1	35.3	51.1	-89.0	-17.6	-6.8
SERV-5	0.0	41.2	54.8	-121.1	-27.9	-17.7

Table 4.8. Combining snapshots prior to compression.

size of snapshots that are combined before compression with snapshots that are compressed and then combined. When using bzip2 or PPMd, precombining the snapshots is very helpful, saving 62% and 42% respectively. However, the three columns on the right of Table 4.8 show that the files resulting from combining snapshots before compression still do not approach the BPC trace compression ratios except in select cases (positive percentages in the table). Combining concrete snapshots from different points in the program requires more temporary storage than individually compressed snapshots because the entire collection would need to be decompressed in order to read a single snapshot.

On average, BPC+PPMd provides a 3.4×, 2.9×, and 2.7× savings over stored predictors compressed with gzip, bzip2, and PPMd respectively. When broken down by workload, the savings of BPC+PPMd over concrete+PPMd ranges from 2.0× (integer) to 5.6× (floating point). Using BPC+PPMd rather than concrete predictors compressed with gzip, bzip2, and PPMd, translates to an absolute savings (assuming 20 traces, 1 billion instructions per trace, and an average of 7.5 instructions per branch) of 257MB, 207MB, and 182MB respectively. Note that this represents the *lower bound* of savings with BPC: if one wishes to study m branch predictors of size $P = \sum_{i=1}^{m} p_i$, the size of the concrete snapshot will grow with mnP, while the BPC trace supports any set of predictors at its current size.

From these results, we note that predictive compressors (bzip2, PPMd, VPC, and BPC) outperform dictionary-based compressors in all cases, often drastically. BPC+bzip2 outper-

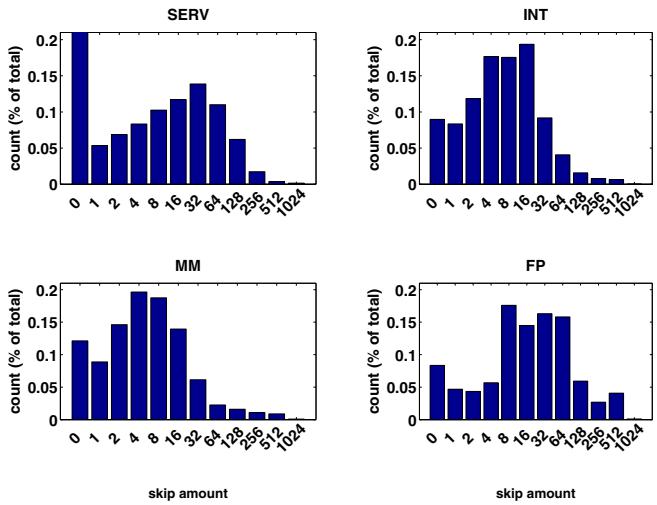


Figure 4-8. Skip amount frequency. Large skip amounts indicate chains of correct predictions. Bucket *i*, labeled *b[i]*, tallies skip amounts greater than or equal to *b[i]*, but less than *b[i+1]*.

forms pure bzip2 in all cases, and BPC+PPMd exceeds stand-alone PPMd compression in 15/20 cases. In a sense, BPC is similar to the Markov modeling used by PPM. However, the additional context (e.g., long global histories and deep return address stack) usually allows BPC to predict better than the simpler model constructed by PPMd. In the cases where PPMd does better, we may be able to tease out additional improvement through the use of stride predictors or improved direction and indirect branch predictors.

Figure 4-8 shows the length of correct prediction chains and helps explain the success of BPC. Recall that long chains are represented by a single number in the *skip amount* output file and a single branch in the *branch record* file. These histograms show the total skip counts for the five traces in each application domain, and we normalize to total number of branches to allow cross-domain comparison. In terms of total branches, we remove over 90% of branches in all cases and we remove over 95% in all but four cases: integer and multimedia are the most troublesome due to lower accuracy in the direction predictor.

4.4.2 Scaling

How can we be certain that the compression ratios observed on this relatively short trace (30 M instructions) will carry through an entire program execution? We extrapolate from the data shown in Figure 4-9 which shows how storage requirements scale over time. Since a single compressed trace suffices to represent all branches in the program, we report the current size of the compressed trace at every 1M instructions. For the concrete snapshots, we report the cumulative snapshot size at the given instant. Note that Figure 4-7 is merely the final data point of Figure 4-9 divided by the number of branches observed and multiplied by 8 bits.

As the program begins, the concrete predictors are largely unused and the easily compressed. Thus, their total size is less than a compressed trace. As the program progresses, the concrete predictors are harder to compress. For all workloads, trace compression scales better than storing concrete predictors. In 15/20 cases, BPC compression fares better than PPMd compression of a raw branch trace; in two other cases (INT2 and FP1) it is competitive or nearly equal, leaving three cases (MM1, MM2, and INT3) in which PPMd outperforms BPC+PPMd. The server benchmarks present an interesting challenge to the compression techniques. In general, these workloads contain branches which are harder to predict and phase changes which are more pronounced. BPC, with its hardware-style internal branch predictors, is more suited to quick adaptation than PPMd which uses more generic prediction. When returning to a phase, BPC's large tables and long history allow better prediction than PPMd, which must adjust its probability models as new inputs are seen; when old inputs return, the model's representation of old data is less optimal. Phase changes are visible as steps in the lines of Figure 4-9; BPC often reacts to phase changes more subtly than PPMd.

The figure shows trends developing in the early stages of execution that should continue throughout the program. A trace compressed with BPC will grow slowly as new static branches appear, but reoccurrence of old branches will be easily predicted and concisely expressed (unless purged from the model). Storage of concrete snapshots grows with mnP as discussed in Section 4.4.1.

4.4.3 Timing

We have shown the storage advantages of trace-based reconstruction versus snapshot-based reconstruction, but we must show that the time required to compress and decompress the data does not outweigh the space savings. In the case of BPC or BPC+general-purpose *compression* for snapshot generation, the cost is negligible. BPC requires simple predictors and tables which add little time to functional simulation. The second-stage, general-purpose compressors (gzip, bzip2, and PPMd) are highly optimized and use fixed-size tables, so they remain fast throughout the compression process. Furthermore, compression is performed once, so the creation time of a microarchitectural snapshot can be amortized over many detailed simulations. We no longer have to guess likely configurations, fix a maximum size, or regenerate a snapshot to reflect a microarchitectural change.

Decompression speed is more important. We presume a parallel methodology in which independent snapshots are produced and used to warm-up state for detailed samples on multiple machines. Each host could simulate a cluster of adjacent samples to amortize the work of seeking to a region in the branch trace. In such a situation, runtime is limited by the time to warm the final sample in program order. When working with a non-random-access compressed trace such as BPC, the warming for the final sample in the program requires examining every branch in the trace. While this is much slower than directly loading a snap-

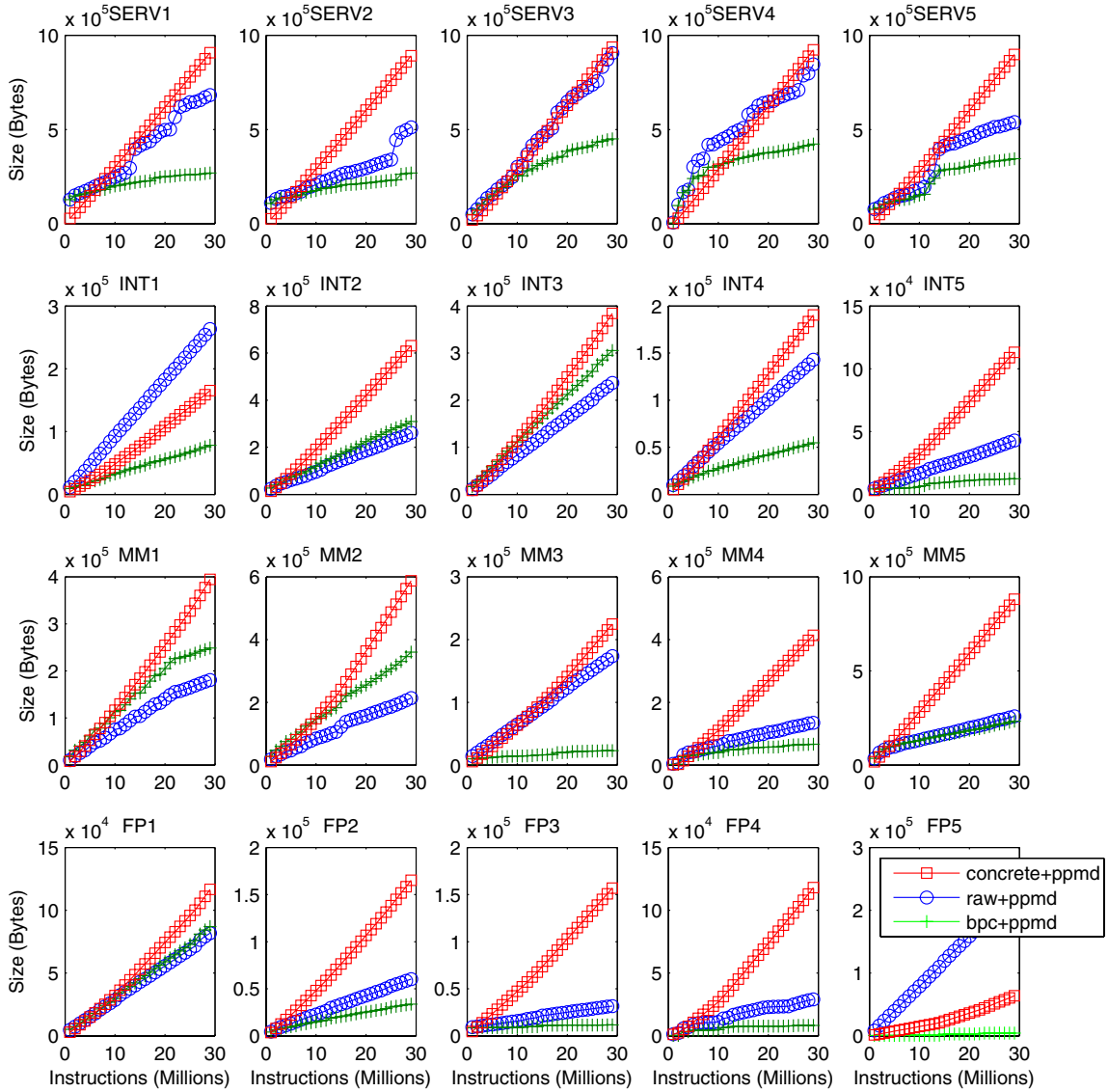


Figure 4-9. BPC storage requirements grow slower than that of concrete snapshots.

	SERV	INT	MM	FP	average
gzip	7.27	17.71	15.68	20.23	13.02
bzip2	0.79	0.67	0.71	0.65	0.70
PPMd	0.81	1.12	1.14	1.30	1.06
VPC+bzip2	1.29	1.90	2.03	2.47	1.82
VPC+PPMd	0.95	1.43	1.46	1.68	1.32
BPC+PPMd	2.23	3.18	2.98	4.10	2.98
sim-bpred		1.09		0.34	0.50

Table 4.9. Performance of BPC and general-purpose decompressors. Table shows millions of branches decompressed per second (harmonic means).

shot of microarchitectural state, it is much faster than functional simulation. Intuitively, warming via branch trace decompression can be faster than functional simulation: not only are there many fewer branches than total instructions, but for each branch, only a few table updates are required rather than an entire decode and execute phase. We have traded some speed for lots of flexibility while remaining several times faster than traditional functional branch predictor warming. Our traces average one branch every 6.42 instructions, and BPC+PPMd can decompress branches at an average rate of 3.245 million branches per second on a 3 GHz host. Thus, BPC adds an average of 48 seconds for every billion instructions on our test platform.

Table 4.9 gives an estimate of the additional time needed to use each decompression scheme. The times were collected on a Pentium 4 running at 3 GHz. BPC, VPC and PPMd were compiled with gcc 3.4.3 -O3, and vendor-installed versions of gzip and bzip2 are used. Timing information is the sum of user and system time reported by `/usr/bin/time`. We require each application to write its data to stdout, but redirect this output to `/dev/null`. For VPC, we modify the generated code so that 2nd-stage compression may be performed in a separate step; the sum reported in the table may be slightly slower than had the 2nd-stage compression been performed inline, but it allows us to examine alternative 2nd-stage compressors. sim-bpred is the branch predictor driver distributed with the popular SimpleScalar toolset [14]. We run SPEC CPU2000 benchmarks using Minnespec datasets ([55]) with sim-bpred's static not-taken predictor to show how quickly a fast functional simulator can decode and identify branch instructions. Note that the SPEC CFP2000 average speed is hurt by several benchmarks with a very small percentage of control instructions (e.g., 50 or 120 instructions per branch); while it is a representative average, it is difficult to compare directly with the CBP traces which have closer to ≈ 15 instructions per branch.

Our original BPC implementations used state-of-the-art perceptron predictors and the standard template library (STL) which dominated runtime. The current implementation, which uses large lookup-table-based predictors and no STL, strikes a balance between speed and compression. The table shows that while the impressive compression ratios observed in Figure 4-7 do not come for free, one can still obtain decompression speeds that surpass an optimized simulator. Not only is BPC faster than a fast RISC functional simulator, we

	min	max	mean	st. dev
SERV	25.1%	42.0%	30.8%	7.2%
INT	2.6%	43.0%	23.4%	18.5%
MM	4.0%	43.9%	27.3%	18.2%
FP	1.8%	20.6%	9.1%	7.7%

Table 4.10. Decompression time is shared between general-purpose decompressor and BPC. Table shows statistics for percent of time spent performing general-purpose decompression.

note that the BPC rate will remain constant while functional simulation becomes slower as support for CISC instructions and full system simulation is added.

We see that PPMd performs better than the more common bzip2 when dealing with all categories of branch traces. Combining BPC with PPMd gives us performance up to 3.9× faster than PPMd alone because BPC acts as a filter allowing PPMd to operate on a smaller, simpler trace. The table also shows VPC times for its default 2nd-stage compressor (bzip2) and VPC combined with PPMd. VPC performs best with bzip2, but appears slower than BPC. The speedup is due to a combination of a BPC's simpler hash function; fewer and smaller predictors which may relieve cache pressure; and a more-easily compressed output to the general-purpose compressor.

The decompression time is dominated by BPC rather than the general-purpose compression phase, but the fraction varies depending on workload. For example, the small, highly compressed floating point branch traces spend as little as 1.75% of decompression time performing general-purpose decompression, while the server traces require at least 25.0%. Table 4.10 shows the percentage of time spent performing general-purpose decompression for each class of trace.

4.4.4 Summary of results

Figure 4-10 summarizes the space and time information from Figure 4-7 and Table 4.9 and is a convenient way to choose the optimal compressor for a particular goal (speed or size) and dataset. For each of four workloads, we plot the average bits-per-branch and speed of decompression for each class of traces. We use harmonic mean for the rate on the y-axis. The most desirable compressors, those that are fast and yield small file sizes, appear in the upper left of the plots. Note that gzip does not appear on the plot: it is the clear winner in speed, but its compression ratio makes it undesirable for snapshots as we saw in Figure 4-7.

For each application domain, BPC+PPMd is the fastest. In terms of bits-per-branch, BPC+PPMd is similar to VPC for highly-compressible floating point traces and similar to PPMd for integer benchmarks. For multimedia, PPMd creates the smallest files, while BPC+PPMd performs significantly better than all its peers for hard-to-predict server benchmarks. High speed and small files across application domains are the strengths of BPC.

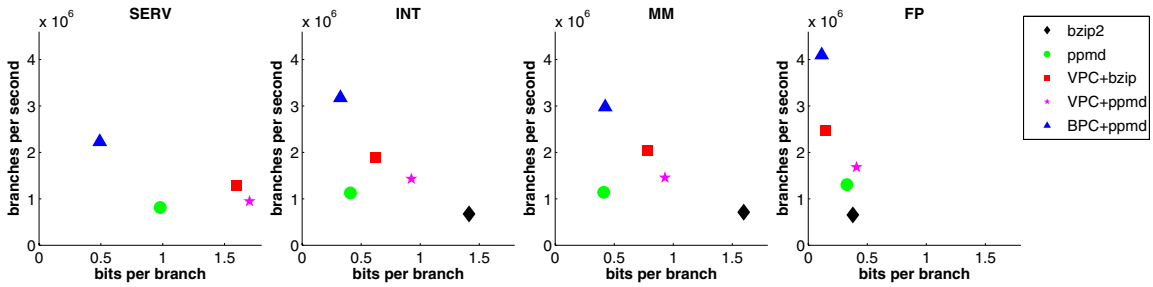


Figure 4-10. Decompression speed vs. Compressed Ratio. The optimal compressed trace format is in the upper left of each plot. Decompression speed across applications is reported with harmonic mean.

4.4.5 Alternative internal predictors

Our choice of BPC's internal predictors was not arbitrary; we searched for the predictors that gave the best compression ratio at the highest speed. Figure 4-11 shows the accuracy of several predictors including several from the first Championship Branch Prediction contest. From left to right we see: a simple gshare predictor with 15 bits of global history; four Alpha 21264-style tournament predictors of varying sizes and index functions; O-GHEL and O-GEHL modified to use and 32-bit arithmetic (faster on a 32-bit host) and larger, fixed-size tables which simplify indexing operations; three Piecewise-Linear predictors [50] (pwl_pow2 uses power-of-two size tables to avoid a modulus operation during indexing and a faster XOR index function which can lead to more collisions and lower accuracy. pwl_big increases the hash table size to reduce collisions); ppm is a CBP contest entry that uses the longest of four available global histories to index into the PHT [73]; ppm_cache and ppm_noshift were optimization attempts that failed to produce noticeable speedup — ppm_noshift increases the number of counters, but uses shorter history and increases accuracy in many cases.

Clearly, O-GEHL [96], (the contest winner) is a highly accurate predictor that would help improve BPC's ability to provide a highly-compressed trace. However, as shown in Figure 4-12, the most accurate predictors can be slow due to their loop-based software implementation. Some of these loops were hard to unroll, but even when I could unroll a loop, the additional work per branch led to a slowdown. Modifications were made to simplify indexing functions and hash table performance, but in the end, we chose a tournament predictor that could be implemented with a handful of table lookups. It appears that ppm_noshift could outperform the chosen predictor with little decrease in speed, but this predictor was not used in the results above.

4.4.6 Comparison to prior work

Value-predictor based compression (VPC) is a recent advance in trace compression [15]. Its underlying predictors (last-n-value, strided, and (differential) finite context) are more general than the branch direction and target predictors found in BPC. As such, VPC has trouble

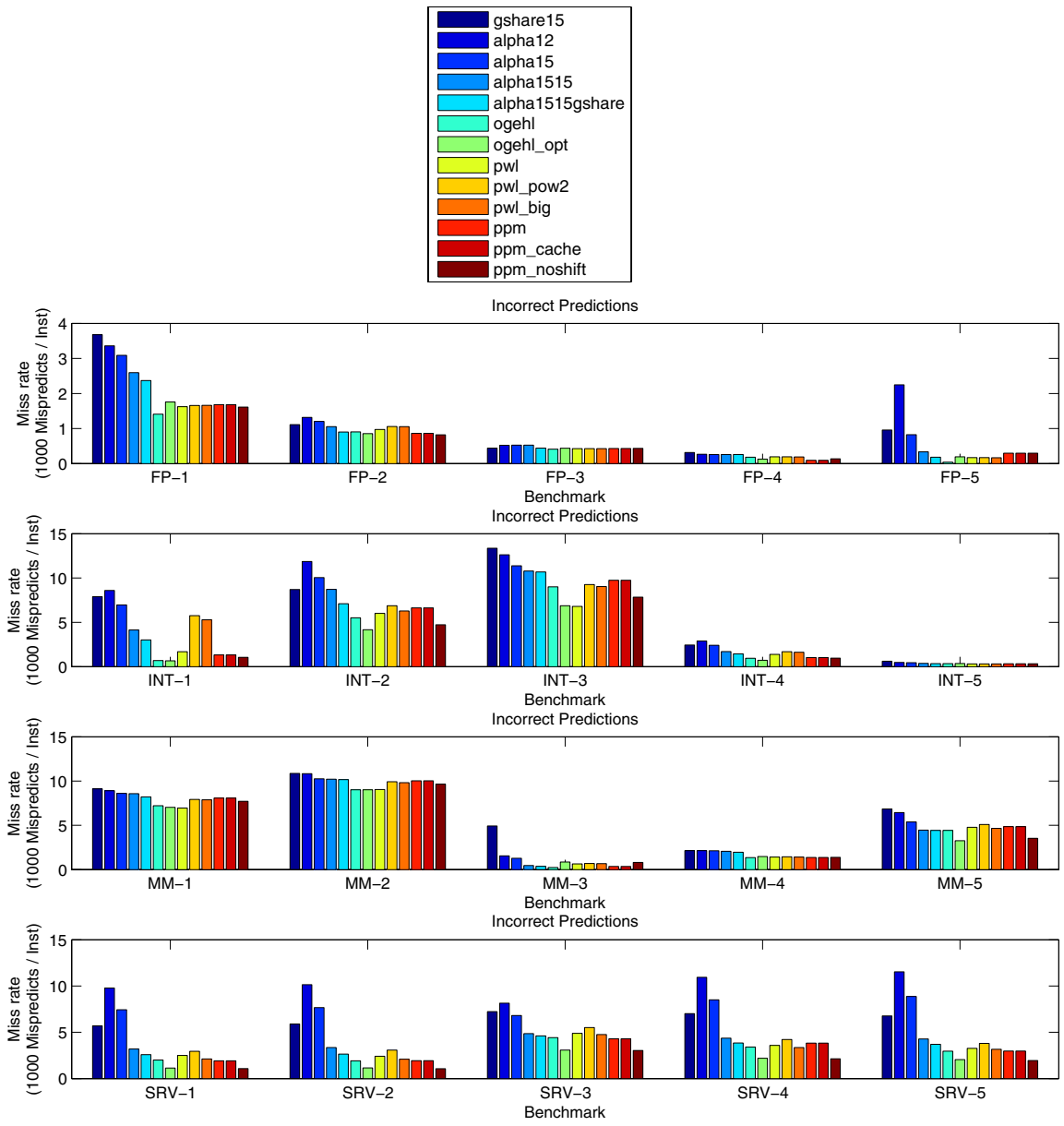


Figure 4-11. Accuracy of several direction predictors.

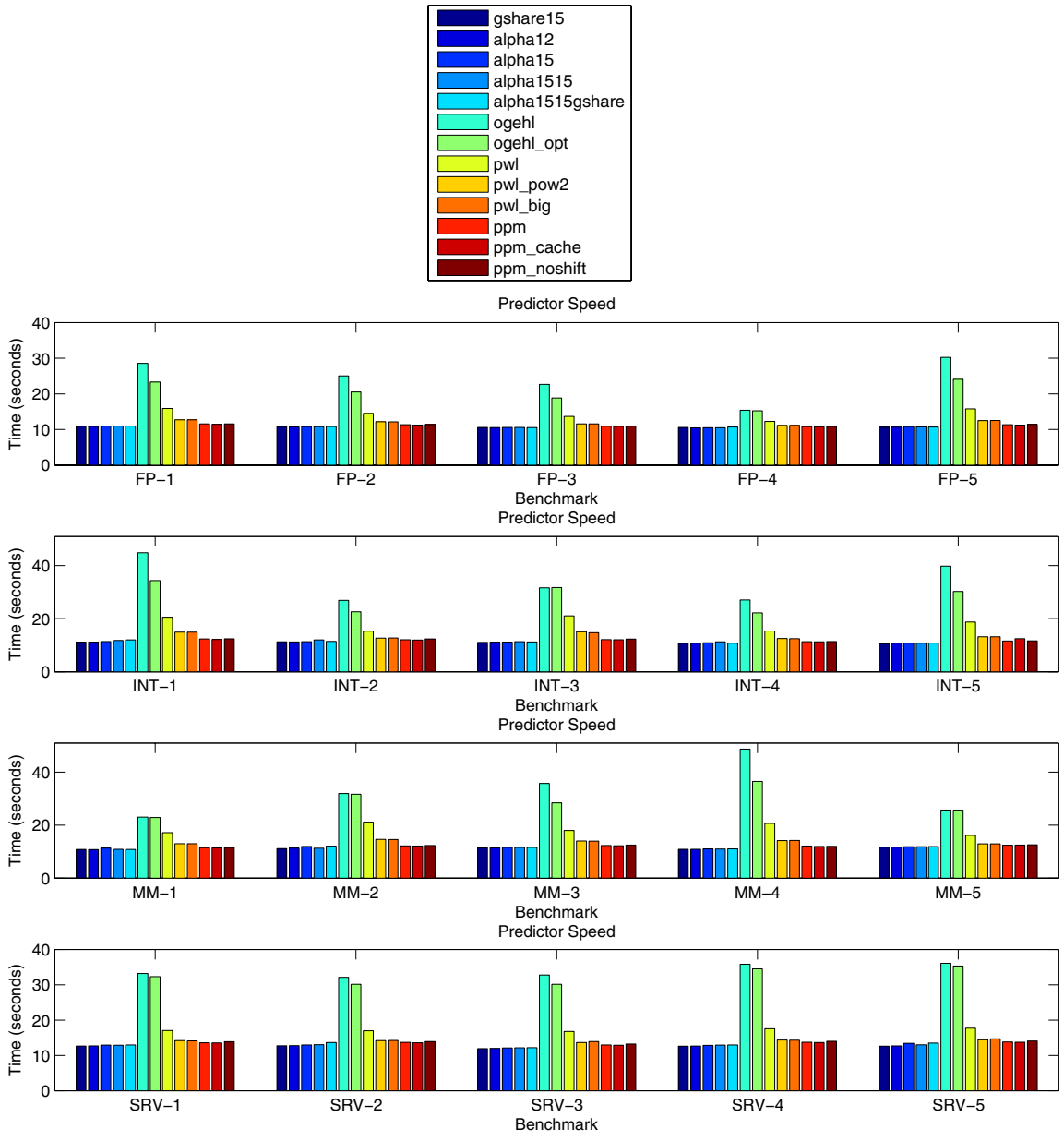


Figure 4-12. Speed of several direction predictors.

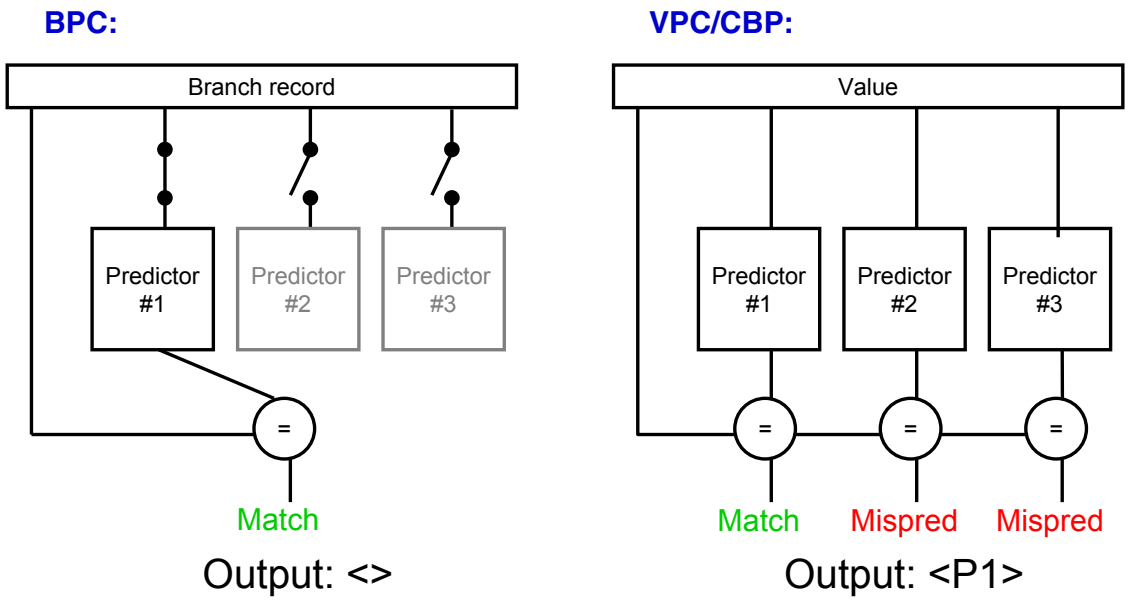


Figure 4-13. Comparing BPC organization with prior work.

with branch traces in which branch outcome may only be predicted given a large context. Both predictors must emit incorrectly-predicted branches, but in contrast with BPC, VPC runs several predictors in parallel and emits a code to indicate which predictors should be consulted for each record. When an internal predictor cannot be used, the unpredictable portion of the record is output. Separate output streams are used corresponding to each internal predictor.

To see the improvement possible with BPC's specialized predictors, we used TCgen, an automated code generator, to generate a VPC compressor/decompressor pair [17]. We begin as suggested by the developers, by generating code with many predictors (we used 44 of different classes and context lengths); running it on our traces; and refining to include only those predictors that perform best. Paring down the predictors eliminates additional output streams and reduces variability in the correct-predictor index that can negatively effect compression. Eventually we settled on the TCgen specification in Figure 4-14 which uses 18 predictors, eliminates the simpler last-value predictor, and uses finite-context predictors only where most useful. Figure 4-7 and Table 4.9 include data for VPC. We see that BPC compresses branch trace data better than VPC in 19/20 cases (all 20 if we always choose the best 2nd-stage compressor for each) and is between 1.1 and 2.2 times faster. We compressed raw VPC streams with both bzip2 and PPMd to show the effect of the second stage compressor. VPC was tuned for integration with bzip2, and this is evident in the results.

The CBP trace reader was written to favor compression ratio over decompression speed and was distributed without excessive tuning [107]. CBP uses a simpler set of predictors: gshare with 14 bits of history, a path-based indirect branch predictor with 2^{10} entries, a 128-

```
TCgen Trace Specification;
0-Bit Header;
32-Bit Field 1 = {L1 = 1, L2 = 131072: DFCM3[2], FCM3[2], FCM1[2]};
32-Bit Field 2 = {L1 = 65536, L2 = 131072: DFCM3[2]};
8-Bit  Field 3 = {L1 = 65536, L2 = 131072: DFCM3[2]};
32-Bit Field 4 = {L1 = 65536, L2 = 131072: DFCM3[2], FCM1[2]};
8-Bit  Field 5 = {L1 = 65536, L2 = 131072: DFCM3[2], FCM1[2]};
PC = Field 1;
```

Figure 4-14. Tuned TCgen specification.

entry return address stack (RAS), a static info cache with 2^{18} entries, and two target caches with a total of $2^8 + 2^{16}$ entries. Like VPC, a code is emitted which describes which predictors are correct. Unlike VPC, the code is followed by a variable-length record that contains only the information that must be corrected. CBP exploits the variable-length nature of x86 instructions. In addition, it includes all instructions, not just branches, so it does not need to encode fall-through instruction addresses. Though it uses similar techniques, a direct comparison with CBP is not possible (CBP obtains near-perfect program counter compression due to the interleaving of non-branch instructions). When perfect PC prediction is possible, CBP+bzip2 outperforms BPC in 10/20 cases, but when perfect prediction is not allowed, BPC produces smaller files. In a sense, CBP does chaining as well but outputs the chain amount in a unary coding. For example, five 0's in a row means that internal predictors suffice to produce the next five branch records. With BPC, we merely output "5". While our encoding is simpler, the CBP encoding can lead to long runs of 0's that are easily compressed.

Figure 4-13 illustrates the differing organizations. BPC statically chooses a predictor while VPC and CBP dynamically choose the best predictor, requiring some non-zero-length output per branch.

In conclusion, the specialized nature of our input data and our exploitation of long runs of correct predictions, allow for an extremely efficient implementation that generally exceeds the performance of more general related work.

4.5 Related Work

Apart from the branch trace compressors, much research has been performed to compress memory traces. Memory traces of most programs exhibit temporal locality. This property can be exploited by removing memory references to the same address that are repeated later in the trace. Agarwal and Huffman survey several techniques that exploit temporal locality [3]. One of these techniques, a *cache filter*, uses a small direct mapped cache to filter out records from the trace. Only those memory accesses which miss in the cache filter are recorded in the trace. A cache filter reduces traces by nearly 90–95% with little or no error. Agarwal and Huffman extend this scheme to take advantage of spatial locality as well. They apply stratified sampling, a technique in which samples are pulled from similar strata rather than

randomly across an entire population. Strata are detected with a structure called a *block filter*. When strata with similar performance characteristics exist, as they do when programs execute runs of accesses to the same region of memory, there is a reduction in the total number of samples needed to achieve a fixed confidence. When the cache filter is used prior to a block filter, traces are reduced by nearly 99% with only 10% error in miss rate estimates. The locality properties of memory traces are harder to exploit for branch trace compression as explained in Section 4.2.

The Mache is a technique that exploits spatial locality for lossless trace compression [93]. It keeps a three-entry cache of the last instruction fetch, data load, and data store. When a new address arrives, the appropriate cache is examined. If the difference between the two addresses is small (e.g., less than $\pm 2^{13}$), it is emitted along with a two-bit reference to the cache element. The size of the cache can be varied, but yields only minor improvement given the impact on speed. Such lossless trace compression was not viewed as a method for reducing cache simulation time as faster techniques (such as: sampling, lossy compression, and single-pass algorithms) exist with little loss in accuracy. However, we advocate this use of lossless compression in the domain of branch predictor simulation (Chapter 4) as there appears to be no faster solution to the problem of a microarchitecture-independent representation of a branch predictor.

Stream Based Compression (SBC) takes advantage of the fact that many programs consist of streams of instructions that are repeatedly executed [74]. Each stream is stored once and identified by a unique index; the remainder of the compressed instruction trace references these indexes. Data references are compressed by storing base address, stride, and repetition count.

The recognition that hardware value predictors are tuned to identify reference patterns led to work in Value Predictor-based Compression (VPC) [16]. VPC employs research in value prediction to accurately predict the next value that will appear in a trace. Value predictors provide the likely result of a computation before that result has actually been established. The pattern recognition abilities of advanced value predictors are useful for improving prefetching by predicting addresses that will be accessed in the future. For example, a value predictor could discover that a program is accessing every 57th cache line and prefetch successive lines automatically. Value predictors can also allow a machine to "exceed the dataflow limit" — speculatively removing true data dependencies inherent to a program, e.g., quickly predicting the result of a long multiply instruction [65]. With VPC, a collection of value predictors implemented in software offers several predictions for each element of a trace. When at least one predictor is correct, the element can be omitted from the trace and replaced by a reference to the correct predictor. When combined with general-purpose compressors, this technique yields tremendous compression and high speeds, improving upon the results of SBC. Our BPC technique is similar in spirit to VPC, but contains optimizations that make it especially suited for representing a branch trace and decompressing that trace quickly.

4.6 Summary

We have presented a technique, BPC, which utilizes software branch prediction structures to produce highly compressed branch traces for use in snapshot-based simulation. Using a popular corpus of branch traces, we show that BPC yields high compression ratios and fast decompression speed. Chaining consecutive correct predictions from accurate software branch predictors, BPC achieves compression rates of 0.12–0.83 bits/branch (depending on workload). This is up to 210× better than gzip, up to 52× better than the best general-purpose compression techniques, and up to 4.4× better than recently published, more general, trace compression techniques. By balancing accuracy with quickness, BPC realizes decompression speeds that exceed functional simulation by an average factor of 3–6×. In the context of snapshot-based simulation, BPC-compressed traces serve as microarchitecture-independent representations of branch predictors. We have shown that this representation can require less space than one which stores just a single concrete predictor configuration, and that it permits the reconstruction of any sort of branch predictor.

Chapter 5

Comparing Experimental Results of Multiprocessor Simulation

A computer architect faces many challenges when simulating a multiprocessor target. Not only is it challenging to build an accurate simulator, but interpreting experimental results becomes more complex in the presence of the non-determinism inherent to multiprocessor systems. This chapter provides a brief list of popular multiprocessor simulators to demonstrate the acceptance of a full-system approach to multiprocessor simulation. Next, we discuss the difficulty of comparing the outcome of two experiments in the presence of multithreading. Finally, we present the problem of variability, which can cause the architect to misinterpret the results of his work. The slowdown introduced by the complexity of full-system multiprocessor simulation can be magnified by the multiple trials required to observe variability. This slowdown helps motivate the work of this thesis.

5.1 Current multiprocessor simulators

Implementing a simulator for a multiprocessor target requires a method for specifying the interleaving of threads of execution. When the number of target processors meets or exceeds the number of threads in a workload, multiple target CPU models can be instantiated, each responsible for running a single thread. When there are more threads than target CPUs, the architect may schedule threads manually to impose some ordering [78]. However, full-system simulation is more realistic, as it allows a simulated operating system to schedule processes subject to events in the target system. Running an operating system in a simulator requires the inclusion of device models such as the disk, realtime clock, interrupt controller, and console. It also requires that the simulated CPU faithfully implement the target ISA to a level sufficient for booting the operating system.

Full-system simulation is used by Stanford SimOS [92], Michigan M5 [11], Wisconsin GEMS [69], and the Carnegie Mellon SimFlex project [124]. The latter two rely on Sim-

ics [68], a commercial product. Full-system simulation also appears in AMD SimNow [1], Intel SoftSDV [117], and IBM Mambo [12]. These tools are used both for internal performance modeling and with third parties for software bring-up. Such full-system simulation is useful even for single-threaded applications on a uniprocessor as it takes into account the performance implications of the operating system and other devices. Because these systems offer so much fidelity, they run extremely slow. A slowdown of 10–100× is common when compared to simpler simulators [12, 124].

5.2 Defining a task

The task whose performance we wish to improve may be difficult to isolate, especially in the presence of multithreaded processors or shared memory. A common task for a two-way simultaneous multithreaded (SMT) processor is executing two independent applications. If one application runs substantially longer than the other, halting simulation at the end of the shorter application will leave us with partially finished work in the longer application. Ending simulation at the completion of the longer application requires that we make a decision about what is now running in place of the short application: does it repeat, does an idle loop take its place, does a new application begin? Even if the two applications include roughly the same amount of instructions, contention for shared resources may handicap the performance of one but not the other, leading to the same complications. Running for a fixed number of instructions, rather than stopping at the end of an individual application, is an even worse choice as it removes the concept of work entirely.

5.2.1 Background

The FAME methodology solves this problem with repetition to generate representative workloads on multithreaded processors [122]. When comparing the instructions per cycle (IPC) of an SMT using two different fetch policies, seemingly reasonable stopping conditions result in reported improvements ranging from 13% to 53%! To narrow the range of results, the authors recommend a profiling run to record the average IPC of each thread when run by itself. During the actual experiment, which runs multiple target threads simultaneously, the workload is extended by repeating each thread until the average IPC of each thread lies within some threshold of its prerecorded individual IPC. At the expense of increased simulation time, this stopping condition ensures that a representative portion of all threads have been executed.

The FAME methodology relies on an assumption that may not hold true for the cooperative workloads hoped to emerge with new multicore designs: that the behavior of an application (its code signature) does not vary when other applications are running on the same processor. This property is needed to gather a “representative trace” and profile its final IPC. The concept of a representative trace is difficult to define for multiprocessors where variability and thread interleaving play a role in determining application behavior. In addition, the profiling run must be repeated when microarchitectural parameters change.

In a single-threaded program, one could use a count of user-mode instructions to define the length of a task. When two targets have executed the same number of user-mode instructions, we know they have performed the same task because there is a one-to-one correspondence between a program's instructions and the work it accomplishes.

When the operating system is simulated, task switching may occur, and one must be careful when counting user-mode instructions that the instructions belong to the program of interest — not other user threads. When the application of interest is multithreaded and running on a target with shared memory, another possibility arises. Imagine multiple worker threads which write to a common word of memory. The shared word must be locked to prevent inconsistency. Whether or not a thread obtains the lock is a function of contention and timing in the system. When spinlocks are used, a thread will continue to execute instructions until it obtains the lock. Even if the program backs-off or yields immediately upon encountering a held lock, it must re-execute instructions to obtain the lock before it can make progress.

To illustrate an extreme case of this point, we ran a microbenchmark which measured locking strategies on a dual processor system. Table 5.1 shows the number of updates per millisecond, changes of lock ownership, and instruction count per thread. Instructions are counted using binary instrumentation. This perturbs the system, slowing it down with instrumentation routines not part of the actual program, but suffices to illustrate the point. In both cases, the benchmark performs the same work — 10 million updates — but switching from a yield to a spinlock causes a 36%-56% increase in instruction count.

Action on failure to lock	updates/msec	owner changes	Thread 1 Insts.	Thread 2 Insts
yield()	17,261	10	320,022,610	320,006,348
spin()	11,946	605,032	499,565,899	438,264,621

Table 5.1. Effect of synchronization overhead on instruction count.

With care, one could determine which instructions are involved in the overhead of synchronization and omit them from the instruction count attributed to a task. Without access to program source code, however, the separation of real work and synchronization overhead can be extremely difficult.

5.2.2 Example

When source code is available, annotations in the application can inform the simulator of task boundaries. Using this application-level knowledge, we show in this section that the use of instruction count to bound individual samples can be problematic in the face of microarchitectural changes. We augment each application in our benchmark suite with a marker that denotes when its initialization phase has been passed. The benchmarks in the NAS Parallel suite generally perform multiple iterations to refine problem solutions, and we add a marker to count each iteration. The web server and file system benchmarks are organized with multiple clients making requests to a server in a loop. Markers are placed at the end of the main

client loop at the point where the request has been serviced. In the Cilk checkers program, a marker denotes the completion of parallel `srch()` functions. `srch()` is the portion of the code that computes the score of a potential move. A toy benchmark, *random* (RA), is included which includes only a loop generating random updates to a fixed-size memory. The memory size of *random* is chosen to fit in an 8x256 KB cache system but not in an 8x16 KB system. OpenMP directives are used to invoke loop iterations in parallel. Each application is first run with a four-way, 16 KB cache in each CPU. Statistics are gathered beginning at the initialization marker, and the application is terminated after a single work marker. The total number of instructions and cycles is recorded and used to guide a second round of simulations on different simulated targets. The second-round target has an 8-way, 256 KB cache per CPU.

Figure 5-1 shows the additional instructions that were induced when using the performance of the system with small caches to bound simulation in the target with large caches. The baseline in this graph is a 256 KB cache system that uses source code annotation to end simulation after one unit of work. Three policies are shown: samples with fixed total instruction count; samples of fixed cycle length; and samples that measure a fixed number of instructions per processor, allowing processors to continue without measurement as needed to complete the work unit. Ideally, a policy will stop the application once it has accomplished a unit of work. Thus, small bars in the figure indicate that we have closely approximated the number of instructions in one unit of application-level work.

Longer running applications that experience slowdown due to misses (such as CG and RA) perform substantially more instructions because the stop condition derived from the system with 16 KB caches overestimates the number of instructions and cycles required by the larger memory system. Figure 5-2(a) shows the effect of these additional instructions and cycles on the reported cache metrics.

When fixed total instruction count is the best policy (FT, CK, and MG), it is likely the application of interest is dominating our measurements (or the interleaved processes unrelated to the benchmark produce few memory references). Miss penalty is the greatest determinant of performance in our idealized processor model. If an application's miss rate with a small cache resembles its miss rate with a large cache, then the application is using a working set that fits within the small cache, and a sample bounded by a fixed cycle count is adequate to provide the accurate measurements (BT and SP).

AP, CG, and RA work best when stopped after a fixed number of instructions per processor. We choose the processor which executed the least instructions, i, per work unit in the 16 KB configuration and measure each processor in the large configuration for only i instructions. This style of bounding can work well when several processors experience many misses in the small configuration that are not present in the large configuration. By bounding the sample with the instruction count of the fastest processor in the small configuration, the measured instructions in the large configuration are the ones performing work related to the identified work unit and little more.

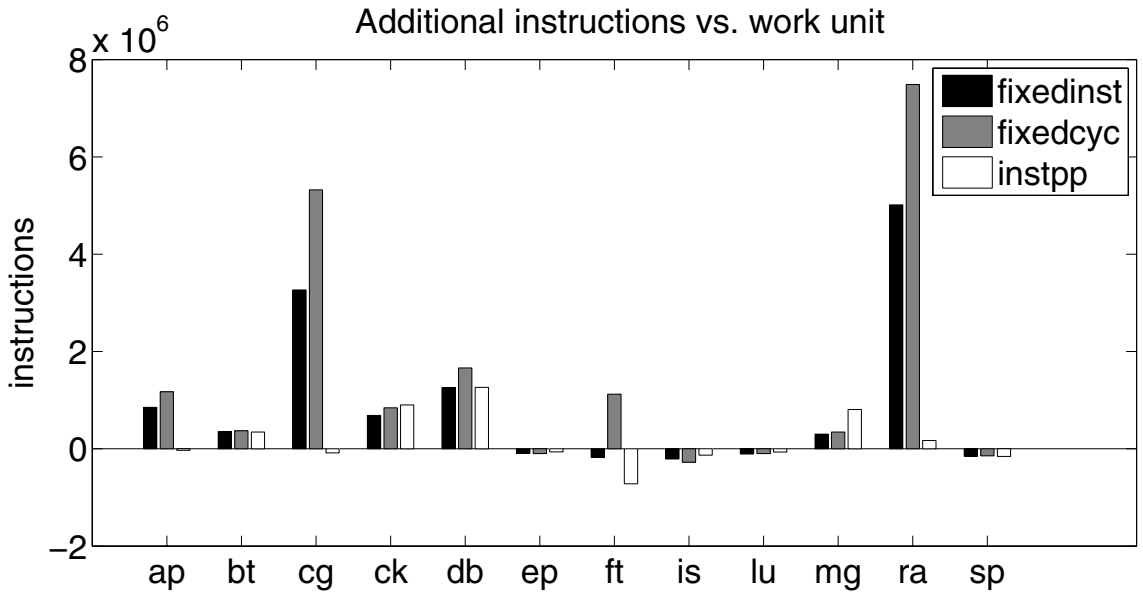


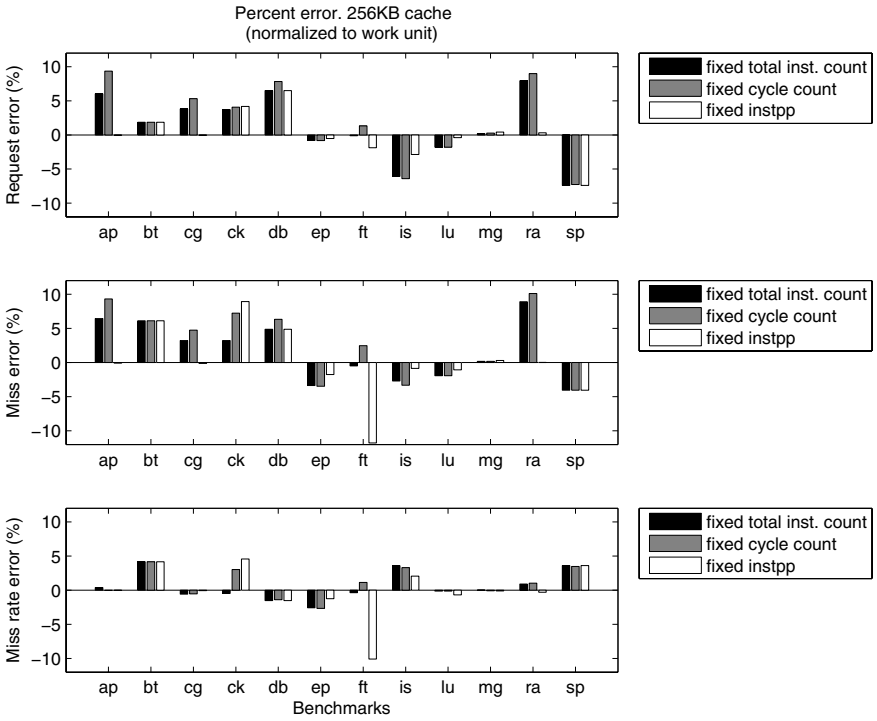
Figure 5-1. Additional execution resulting from incorrect sample bounds.

Figure 5-2(b) is identical to Figure 5-2(a) except that miss penalty has been doubled; we call this configuration *large_slow*. This has a pronounced effect both on the size of error and which technique is best to bound the error. Miss error is magnified for CK and FT, while request error is generally lower. Error for our toy RA benchmark is significantly lower for the fixed instruction and fixed cycle approaches, yet the instruction per processor approach that worked so well with the small cache is no longer useful. A fixed cycle count is now the best choice for AP, BT, and CG as these applications have cycle counts that match closely in small and *large_slow* configurations.

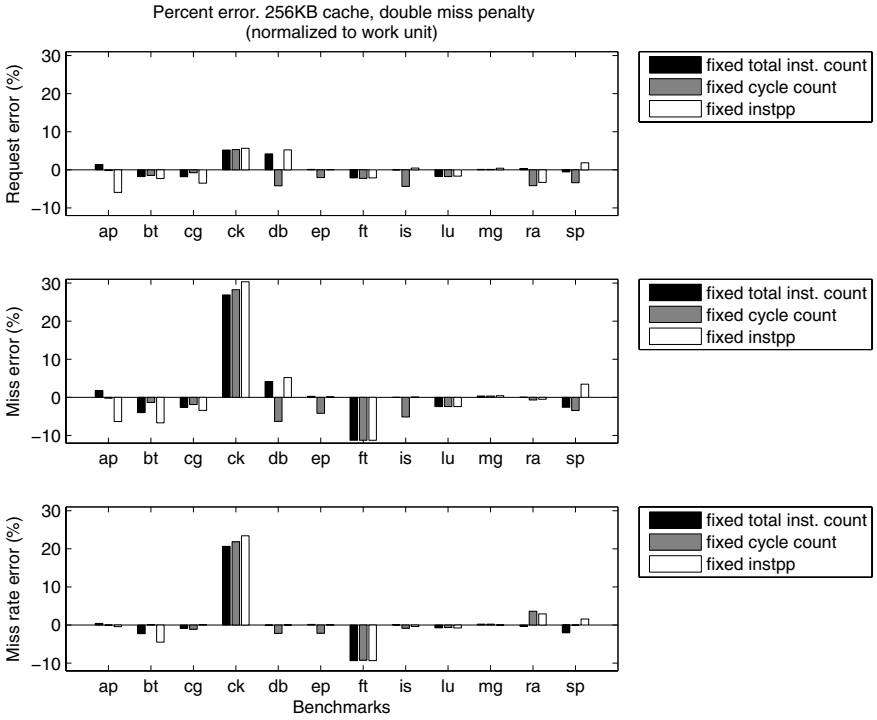
Figures 5-2(a) and 5-2(b) show the difficulty of comparing the performance of two targets when tasks are bounded with microarchitecture-dependent statistics. However, a random sample is equally likely to capture the beginning of a work unit as the middle or end. Thus, a sufficient number of independent samples taken at random should do a good job capturing the performance of an entire task. However, when few samples are used and no effort is made to correlate work with samples, one should use caution when comparing one experiment to another.

5.3 Full-system variability

When system effects are present, there is no longer one correct timing outcome. A full-system simulator allows one to run commercial workloads, but the determinism of a simulation environment does not match the real-world of computer systems. In real systems, I/O may arrive from a network card at random, interrupting the task at hand. Excessive environmental temperature may trigger clock throttling, reducing the amount of work a process completes



(a) Fast cache



(b) Slow cache

Figure 5-2. Sample bounds effect on cache metrics.

before being context-swapped. A hard disk controller may encounter a media error, and the time spent to correct from the error could lead to a thread losing a race for a lock that it typically would win.

5.3.1 Variation in hardware

Awareness of variation is relevant beyond the world of simulation. Much effort goes into determining and eliminating the sources of variation at supercomputing sites [56, 86, 102]. Sometimes this variation may be caused by events as mundane as the periodic interrupts an operating system uses to provide process scheduling and pre-emption. A tick-less operating system has been proposed to avoid the effects of variation in bulk synchronous scientific applications [115]. In these applications, a delay in a single processor is experienced by all processors as they must wait at a barrier before proceeding. Other approaches include reserving a processor-per-SMP-node to handle operating system daemon tasks [87]. The IBM Blue Gene/L has a heterogeneous environment in which certain nodes are devoted to running a full operating system kernel (Linux) that handles communication, file system, paging, etc. Other nodes are compute-only, running a single-process kernel and a user-level runtime library to handle communication with other nodes [113]. The separation is intended to allow full resource utilization by the compute nodes, moving the more variable system-level tasks to the I/O nodes.

5.3.2 Modeling variation in software

Since "correctness" in a shared memory system is defined by the programmer-visible memory consistency model, multiprocessors can run the sample multithreaded program in many correct ways. For example, a computer that presents a *sequentially consistent* memory to the programmer agrees that a given processor's loads and stores will not occur out of program order, but no promises are made with respect to how that processor's memory accesses are interleaved with accesses by other processors. The only guarantee is that all processors will see the same order of requests. This presents a performance modeling challenge: how does a designer know that the performance measured in one simulation is the performance that will be realized during another simulation with a different memory access order?

Microarchitectural changes have a straightforward effect on uniprocessors, but may introduce surprising behavior on multiprocessors. To demonstrate this effect, we can use timelines such the one shown in Figure 5-3. Time (measured in cycles) runs from left to right. Each processor is represented by a line in the plot, and each < color, marker shape > pair represents a distinct address space. Cooperating threads have the same color and marker shape. The plot is initiated by an annotation in the application and ends when the application indicates a completed work unit. Each subplot shows the same amount of work. Address space changes are noted by instrumenting the simulator to report changes to the x86 CR3 page table base register. The three subplots correspond to an eight-processor system with

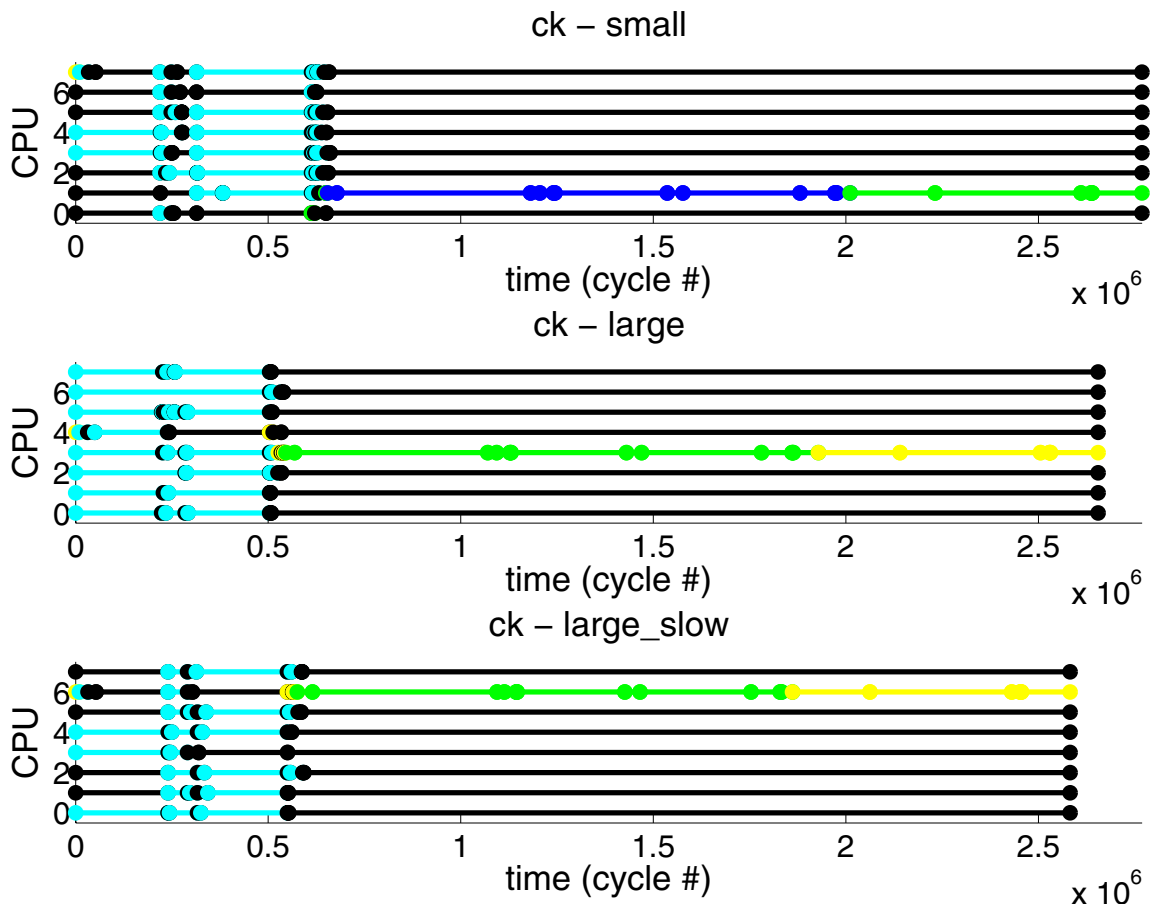


Figure 5-3. Timeline for Cilk checkers (CK) showing effect of microarchitecture on active threads.

16 KB caches, 256 KB caches, and 256 KB caches with double miss penalty. The colors are not necessarily consistent across the three subplots but do correlate for the most part.

Figure 5-3 shows how changes to the microarchitecture can affect the behavior of a multithreaded application. We see that in the large configurations, the Cilk checkers benchmark requires fewer cycles to perform the same amount of work as there are fewer lengthy cache misses. Surprisingly, the *large_slow* configuration requires less cycles than the *large* system, even though *large_slow* requires twice as many cycles to satisfy a cache miss. The unexpected performance improvement is likely due to a different thread interleaving between cycles 0 and 500,000 that allowed the overlapping of useful threads which had been serialized in the large configuration.

Doubling the cache miss penalty was a drastic way to create an example, but even a minor variation in system timing can result in many different observable execution interleavings and lead to vastly different performance estimates as the application proceeds down different paths depending on the order of memory accesses. This phenomenon is called *space variability* [5]. To make useful performance estimates in the face of space variability, one

may conduct multiple runs of the same application, each with slight timing variations. If we assume that the population of all possible timing outcomes has a normal distribution, the sampled runs form a collection of results which can be used to statistically bound the performance prediction. However, the need to repeat simulations increases the number of host CPU-hours required to accurately model a system. While many populations have a normal distribution, the assumption that the distribution of all timing outcomes is normal remains unproven.

We reprise the results of Wood and Alameldeen ([5]) using the benchmarks and multiprocessor infrastructure described in Chapter 3. While Wood and Alameldeen used several runs of an entire benchmark as a sample, we add intra-benchmark random sampling to create a second dimension of sampling. Figure 5-4 shows the effects of both space variation across runs of the same benchmark and the effects of random sampling within a benchmark. The space variation is introduced by varying processor execution order during a window of 8 million instructions. For every 8 million instructions, we choose the fraction of the window used by each CPU and randomize the order in which each CPU ticks. In each graph, we estimate the average cache miss rate across all CPUs. Error bars show a 95% confidence interval as determined from the variance of the intra-benchmark samples. We see, as expected, that more samples narrow the intervals. However, multiple runs of the same application can result in vastly different miss rate estimations. For example, our use of the same random seeds for each benchmark resulted in one run of each benchmark always experiencing a thread interleaving that results in high miss rates. Most importantly, run-to-run difference in estimated miss rate is not purely due to random sampling or else each run's miss rate would be contained within the error bars of its companions.

While this random perturbation of processor speeds is unlikely in a balanced system, it serves to demonstrate that we cannot ignore non-determinism when performing research or developing new multiprocessors. In a software simulation environment, the multiple runs needed to establish the variance of results further increases the latency of an already complicated simulation and suggests the need for fast execution-driven simulation of multiple likely interleavings rather than a single simulation.

In the context of checkpoint-based sampling, capturing variation can be difficult because the checkpoint typically represents a single ordering of memory accesses prior to that point. Once the detailed sampling begins from checkpointed state, timing variation may induce representative space variation. However, it has recently been shown that several million instructions must be executed with a detailed simulator to observe two uncorrelated outcomes for certain benchmarks [124]. Since the creation of confidence intervals based on simple random sampling relies on each sample being independent, it would appear that sampling across several executions with timing variation can not always reliably quantify space variation.

However, for certain benchmarks, it may be appropriate to use the fixed thread interleaving of a checkpoint to begin detailed simulations [124]. Consider a transaction process-

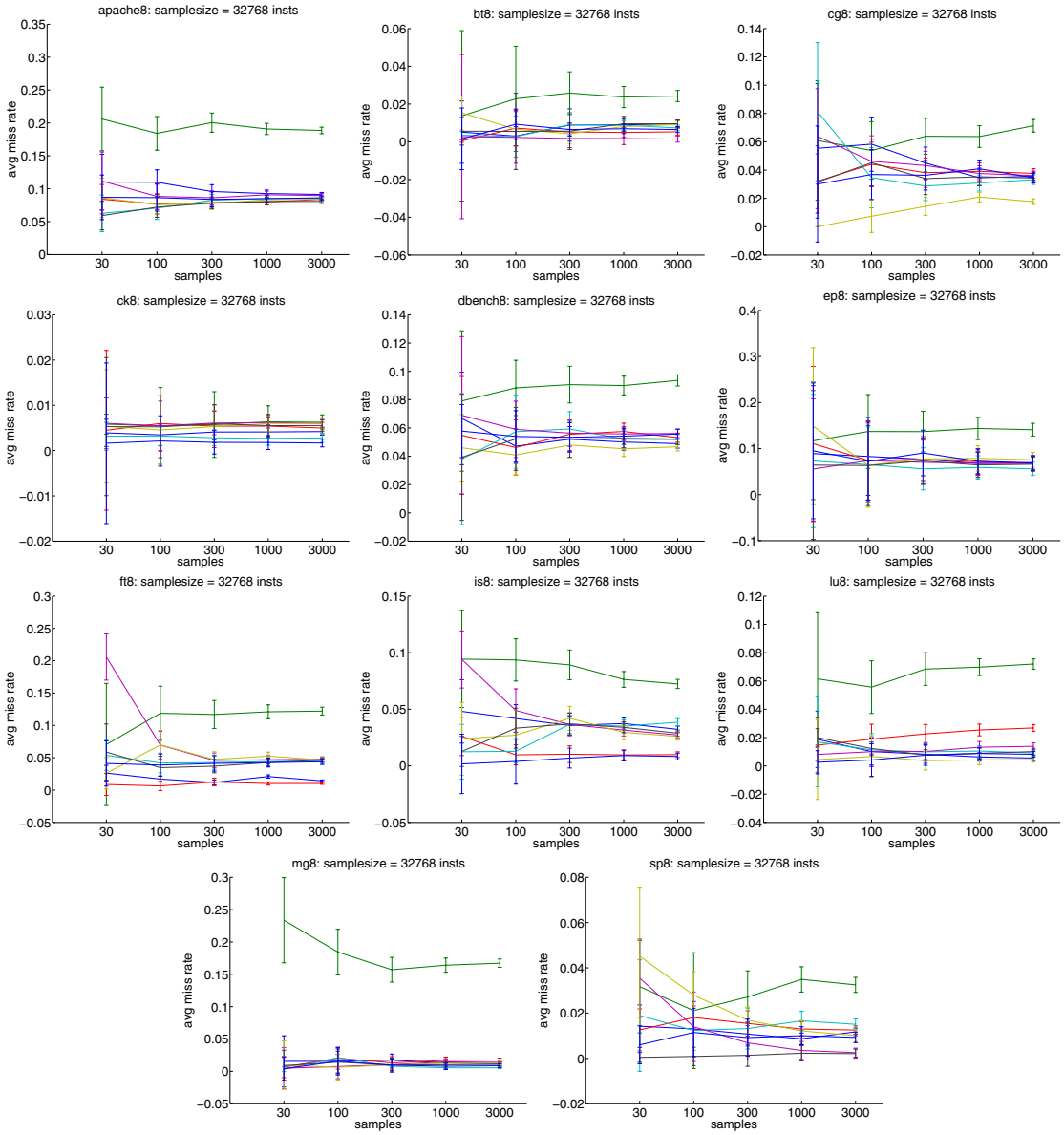


Figure 5-4. Performance estimates differ when system variation is included. Lines represent eight runs, each with different timing.

ing workload with randomly arriving transactions. If enough samples are gathered (about 30 seconds of real time), multiple interleavings will be seen in functional mode with the same probability as in detailed mode. This means that performance estimated from the functionally-generated samples is likely to match performance of a fully detailed simulation with execution-driven thread interleavings.

While checkpoint-based initialization of fixed-instruction samples has been shown to be appropriate for capturing space variation in certain cases, it is not clear that it may be used for all benchmarks. Generating a set of MINSnaps that captures a representative sample of memory access interleavings remains a challenge.

Chapter 6

Conclusion

Simulation is the cornerstone of computer architecture research and development. Simulation in software allows the architect to explore a design space, refine an existing feature, or validate proposed implementations without the expense of creating hardware prototypes. However, as computer systems become more complex, the time required to simulate these systems is growing.

Sampling is a popular approach to reduce simulation time while retaining high accuracy. By confining detailed simulation to a collection of sample points, rather than simulating an entire benchmark, architects are able to get accurate performance results quickly. We can advance to a sample point using a functional simulator that executes a benchmark on a target but does no performance modeling. Checkpoints, stored files that contain target state, are another technique for initializing a sample point. In both cases, it is important that microarchitectural structures in the target contain recent, valid data before beginning performance analysis. Otherwise, the simulation will predict poor performance as it is using empty caches and untrained predictors. To increase accuracy, prior work has investigated ways to avoid this "cold-start" problem.

At one extreme, we can start with a small amount of architectural state and perform detailed simulation to ensure structures like caches and branch predictors are warmed-up. When the structures are warmed, measurement can be enabled. Unfortunately, large caches and complex branch predictors require extensive detailed simulation before they are warmed-up and ready to use.

Alternatively, we can initialize the target at the sample point with microarchitectural state gathered online or stored as a checkpoint. If we have checkpointed the configuration we wish to study, we can warm the microarchitecture immediately. Now, only short-lived structures such as pipeline state and buffers require slow detailed warming. The downside to this approach is increased checkpoint size and the need to know in advance the microarchitectural details that must be warmed.

6.1 Summary of contributions

This thesis has presented new structures that store the state of a memory system and branch predictor in a microarchitecture-independent fashion. These structures, called MINSnaps, offer the benefits of both architecture-only and microarchitecture-dependent snapshots. A MINSnap is nearly as versatile as loading architectural state and performing lengthy detailed warming, and it is nearly as fast as initializing the microarchitecture from a concrete checkpoint.

We were among the first to propose and evaluate functional warming for multiprocessors. We proposed the Memory Timestamp Record, which allows for fast functional warming during online sampling and provides a concise microarchitecture-independent snapshot for use in checkpoint-based sampling. The microarchitecture-independent contents of the MTR are interpreted with a reconstruction algorithm to fill in the state of a particular target memory system.

With Branch Predictor-based Compression, we extended our MINSnap's contents to support branch predictor warming. BPC is a specialization of Value Predictor-based Compression especially suited for branch traces. A branch trace compressed with BPC may be decompressed in less time than functionally simulating each branch. In addition, the use of BPC requires less size than storing multiple microarchitecture-specific branch predictor snapshots per sample point.

6.2 Open problems

Our experiences with the MTR and BPC have been encouraging. Nevertheless, the work suggests several areas for improvement, expansion, and further investigation.

6.2.1 A true MINSnap for branch predictors

BPC transformed the problem of microarchitecture-independent snapshots into a problem of efficient trace compression. Because a BPC-based simulation is, at its heart, a trace-driven simulation, it may be insufficient in some settings. For long-running benchmarks, the inability to perform random accesses into the compressed trace means that to measure the last sample (in program order), one must first decompress the entire trace prior to the sample point. Therefore, the time required to simulate each sample point depends on its location in the program. While this does not preclude independent simulation of each sample point, the imbalance of times makes BPC unattractive for very long programs. The speed problem may be avoided if one is willing to sacrifice complete microarchitecture-independence and space: simply revert to storing a fixed collection of branch predictor snapshots. A compromise is to restrict the analysis to branch predictors that can be represented by a MINSnap, such as those with a single global history register of constrained size, a single pattern history table,

and fixed counter width. For such predictors, a timestamp approach may be successful as illustrated in Section 4.2. Other styles of branch predictors may lend themselves to different MINSnap formats; the discovery of such formats would be a valuable contribution to the field.

6.2.2 Tuning BPC's internal predictors

Our initial implementation of BPC chose its internal predictors to balance accuracy with speed. We also demonstrated, using several predictors from a recent branch prediction competition, that there are other reasonable internal predictors. Further tuning of BPC will always be possible as advances are made in the area of branch prediction.

6.2.3 Quantifying synchronization overhead

The number of instructions that comprise a unit of work varies with system effects and microarchitectural parameters. One source of this error is due to intervening threads. Another source is synchronization overhead. When source code is not available, synchronization code signatures may be used to identify instructions related to acquiring a lock or waiting at a barrier. For instance, if entrance to a critical region is always implemented with a load-link/store-conditional pair followed by a conditional branch to the load-link instruction, repeated instances of these instructions represent overhead due to lock contention. Such clues have been used in hardware mechanisms to discover dynamically occurring parallelism that is hidden by the use of locks [89]. In the context of a software simulator, more resources can be used to analyze instructions and determine whether they are a component of synchronization. However, when the implementation of a synchronization primitive is unknown, when the use of an atomic instruction is ambiguous, or when synchronization behavior is less regular, identification can be challenging [19]. A move toward systems that implement critical sections with transactions may make the identification of overhead an easier task. If work-unit-sized samples are desired with benchmarks compiled to current ISAs, effort is still needed to reliably identify overhead.

6.2.4 Obtaining likely thread interleavings

For workloads that do not exhibit all likely thread interleavings during snapshot generation, it is helpful to induce various thread interleavings to observe a range of performance for a target. In Chapter 3, we experimented with a processor slowdown technique, but the effects of processor slowdown were more pronounced during functional simulation than detailed simulation. The resulting interleavings are not necessarily those likely to be observed in hardware. In Chapter 5, we discussed related work that varied DRAM timing to demonstrate and evaluate space variability. We also showed results from an experiment that chose a different processor tick order and utilization frequency every 8 million instructions. While these schemes effectively produce space variability on full-system simulators, it is not likely

that samples obtained in this fashion represent an independent selection from the population of all possible thread interleavings. If indeed we are not choosing independent samples from the entire population, it is inappropriate to use sample variance to establish confidence intervals. Research into methods for producing appropriate samples via timing variation would be valuable as it would allow these techniques to be applied without making questionable assumptions about the population of possible performance outcomes.

6.2.5 Assessing non-sampling bias on modern microarchitectures

Non-sampling bias, the error due to sampling incorrectly warmed microarchitectural structures, can be made small by careful warming of large structures (caches and branch predictors). We have reviewed existing techniques from the field of trace sampling and more modern proposals such as MRRL, functional warming, and the MINSnaps described in this thesis. These techniques are subject to the effects of wrong-path execution. Wrong-path execution slightly biases results as wrong-path instructions are not usually included in the warming process. Run-ahead processing is a relatively recent microarchitecture proposal that suggests a new form of prefetching [77]. A run-ahead processor is allowed to continue fetching and executing instructions when it would normally be stalled by a cache miss. When the data arrives, some or all of the speculatively issued instructions are re-executed. The goal is that accesses after the miss are allowed to issue to the memory system. When these instructions are re-executed in a non-speculative mode, they may now hit in the cache as it has been primed by these accesses when they first appeared during run-ahead mode. To account for such effects in a sampling-based simulation of run-ahead processors, one could extend the functional simulator to include those accesses occurring during run-ahead mode. However, this slows snapshot generation and ties the snapshot to the run-ahead microarchitecture that generated the snapshot. Quantifying the effects of such accesses on non-sampling bias would be an important step for those seeking to use sampling-based simulation of run-ahead processors.

6.2.6 Combining MINSnaps and hardware-assisted simulation

We have discussed the SimSnap project which used hardware to create snapshots for software simulation. Even as FPGA-based simulation, such as the RAMP project, becomes more popular, it is possible that FPGAs may be used in concert with software simulation. Such a pairing could provide fast generation of snapshots using hardware, allowing complex and observable software simulation of samples initialized from these snapshots. Also, development of timing models is easier in a high level language than it is in a synthesizable hardware description language. Thus, one may wish to resort to such software models while hardware is being developed or as a means to guide the creation of hardware models. While an FPGA may be reconfigured to produce state for any microarchitecture, it may be more desirable to implement MINSnaps in hardware to reduce the amount of state that needs to be stored

and later processed in software. For memory system MINSnaps, it may be necessary to fix a homogeneous and maximum cache size and use a stack-based approach in order to fit in the resources of an FPGA.

If industry adopts hardware extensions such as Dynamic Instruction Stream Editing (DISE) [27], MINSnaps could be generated at the hardware level by creating short instrumentation routines performed on every load, store, and branch. This is similar in spirit to building tracing support into hardware via microcode patches [4]. However, microcode has been relegated to supporting legacy instructions, and DISE represents a modern, low-impact approach to dynamic hardware-based instrumentation.

6.3 Concluding remarks

This thesis has presented the Memory Timestamp Record (MTR) and Branch Predictor-based Compression (BPC). Together, these techniques support multiprocessor simulation efforts that rely on sampling. By summarizing the effects of multiprocessor program execution in a microarchitecture-independent snapshot, MTR and BPC allow the researcher to amortize the time spent generating the snapshot across multiple experimental targets — even when each target represents a new microarchitecture. This reusability lowers storage requirements and speeds simulation, reducing the time required to design, evaluate, and refine computer architectures.

Bibliography

[1] Advanced Micro Devices. *AMD SimNow™Simulator 4.0.0 User's Manual, Revision 1.41*, Jan. 2006.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4), Nov 1988.

[3] A. Agarwal and M. Huffman. Blocking: exploiting spatial locality for trace compaction. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–57, 1990.

[4] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *International Symposium on Computer Architecture*, June 1986.

[5] A. R. Alameldeen and D. A. Wood. Addressing workload variability in architectural simulations. In *International Symposium on High Performance Computer Architecture*, Feb. 2003.

[6] Arvind, K. Asanović, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. RAMP: Research accelerator for multiple processors – a community vision for a shared experimental parallel HW/SW platform. Technical Report CSD-05-1412, EECS Department, University of California, Berkeley, 2005.

[7] K. C. Barr and K. Asanović. Branch trace compression for snapshot-based simulation. In *International Symposium on Performance Analysis of Systems and Software*, Mar. 2006.

[8] K. C. Barr, R. Matas-Navarro, C. Weaver, T. Juan, and J. Emer. Simulating a chip multiprocessor with a symmetric multiprocessor. In *Boston Area Architecture Workshop*, Jan. 2005.

[9] K. C. Barr, H. Pan, M. Zhang, and K. Asanović. Accelerating a multiprocessor simulation with a memory timestamp record. In *International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

[10] F. Bellard. *QEMU Internals*, 2006. `http://fabrice.bellard.free.fr/qemu/qemu-tech.html`.

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

[12] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[13] G. Bronevetsky, M. Schulz, P. Szwed, S. Zaman, and K. Pingali. Application-level checkpointing for shared memory programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[14] 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.

[15] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The VPC trace-compression algorithms. *IEEE Transactions on Computers*, 54(11), Nov. 2005.

[16] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[17] M. Burtscher and N. B. Sam. Automatic generation of high-performance trace compressors. In *International Symposium on Code Generation and Optimization*, Mar. 2005.

[18] L. Ceze et al. Full circle: Simulating Linux clusters on Linux clusters. In *Fourth LCI International Conference on Linux Clusters: The HPC Revolution*, June 2003.

[19] J. Chang and X. Wang. Lock behavior characterization of commercial workloads. Unpublished University of Wisconsin CS/ECE 757 Project Report, May 2002.

[20] D. Chiou, H. Sunjeliwala, D. Sunwoo, J. Xu, and N. Patil. FPGA-based fast, cycle-accurate, full-system simulators. In *2nd Workshop on Architecture Research using FPGA Platforms*, 2006.

[21] E. S. Chung, J. C. Hoe, and B. Falsafi. Protoflex: Co-simulation for component-wise FPGA emulator development. In *2nd Workshop on Architecture Research using FPGA Platforms*, 2006.

[22] D. W. Clark and J. S. Emer. A characterization of processor performance of the VAX-11/780. In *International Symposium on Microarchitecture*, June 1984.

[23] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4), Apr. 1984.

[24] Condor Team, University of Wisconsin-Madison. *Condor®Version 6.7.19 Manual*, May 2006.

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

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

[27] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *International Symposium on Computer Architecture*, June 2003.

[28] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *ACM SIGMETRICS conference on measurement and modeling of computer systems*, 1988.

[29] R. Covington et al. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[30] J. D. Davis, C. Fu, and J. Laudon. The RASE (rapid, accurate simulation environment) for chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 33(4):14–23, 2005.

[31] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *International Symposium on Computer Architecture*, July 1998.

[32] K. Driesen and U. Hölzle. The cascaded predictor: Economical and adaptive branch target prediction. In *International Symposium on Microarchitecture*, Dec. 1998.

[33] P. Dubey and R. Nair. Profile-driven sampled trace generation. Technical Report RC 20041, IBM Research Division, Apr. 1995.

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

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

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

[37] J. Emer, Intel Corporation. Personal communication, Nov. 2004.

[38] J. Engblom and C. M. Holm. A fully virtual multi-node 1553 bus computer. In *Data Systems in Aerospace*, May 2006.

[39] J. K. Flanagan, B. E. Nelson, J. K. Archibald, and K. Grimsrud. BACH: BYU Address Collection Hardware, the collection of complete traces. In *International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, 1992.

[40] J. Gailly and M. Adler. gzip 1.3.3. `http://www.gzip.org`, 2002.

[41] G. Ganapathy, R. Narayan, G. Jorden, D. Fernandez, M. Wang, and J. Nishimura. Hardware emulation for functional verification of K5. In *Design Automation Conference*, 1996.

[42] J. Gateley, M. Blatt, D. Chen, S. Cooke, P. Desai, M. Doreswamy, M. Elgood, G. Feierbach, T. Goldsbury, and D. Greenley et al. UltraSPARC-I. In *Design Automation Conference*, 1995.

[43] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):31–35, March 2004.

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

[45] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[46] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[47] J. Hong, E. Nurvitadhi, and S.-L. L. Lu. Design, implementation, and verification of active cache emulator (ACE). In *International Symposium on Field Programmable Gate Arrays*, 2006.

[48] V. Iyengar, L. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Symposium on High Performance Computer Architecture*, Feb. 1996.

[49] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[50] D. A. Jiménez. Idealized piecewise linear branch prediction. In *Championship Branch Prediction Competition*, 2004.

[51] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *International Symposium on High Performance Computer Architecture*, Jan. 2001.

[52] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, Oct. 1999.

[53] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2), March-April 1999.

[54] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 1994.

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

[56] W. Kramer and C. Ryan. Performance variability of highly parallel architectures. In *International Conference on Computational Science*, May 2003.

[57] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *IEEE Workshop on Workload Characterization*, Sept. 2000.

[58] S. Laha, J. A. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, Feb 1988.

[59] F. Larsson, P. Magnusson, and B. Werner. Simgen: Development of efficient instruction set simulators. Technical Report R97-03, Swedish Institute of Computer Science, 1, 1997.

[60] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *24th Annual International symposium on Computer Architecture*, Jun 1997.

[61] G. Lauterbach. Accelerating architectural simulation by parallel execution. In *Hawaii International Conference on System Sciences*, Jan. 1993.

[62] K. Lawton et al. Bochs. `http://bochs.sourceforge.net`.

[63] C. Leiserson et al. Cilk 5.3.2. `http://supertech.lcs.mit.edu/cilk`, June 2000.

[64] K. M. Lepak, H. W. T. Cain, and M. H. Lipasti. Redeeming IPC as a performance metric for multithreaded programs. In *International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[65] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, Dec. 1996.

[66] G. Loh. Revisiting the performance impact of branch predictor latencies. In *International Symposium on Performance Analysis of Software and Systems*, Mar. 2006.

[67] 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*, June 2005.

[68] P. S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[69] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset,. In *Computer Architecture News*, Sept. 2005.

[70] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[71] E. M. McCreight. The Dragon computer system, an early overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[72] S. McFarling. Combining branch predictors. Technical Report WRL Technical Note TN-36, Digital Equipment Corporation, 1993.

[73] P. Michaud. A PPM-like, tag-based predictor. In *Championship Branch Prediction Competition*, 2004.

[74] A. Milenkovic and M. Milenkovic. Exploiting streams in instruction and data address trace compression. In *IEEE Workshop on Workload Characterization*, Oct. 2003.

[75] M. Milenkovic, A. Milenkovic, and J. Kulick. Demystifying Intel branch predictors. In *Workshop on Duplicating, Deconstructing and Debunking*, May 2002.

[76] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A fast and portable architecture simulator. In *Workshop on Performance Analysis and its Impact on Design*, June 1997.

[77] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *International Symposium on High Performance Computer Architecture*, Feb. 2003.

[78] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *International Conference on Computer Design*, 1996.

[79] S. Nussbaum and J. E. Smith. Statistical simulation of symmetric multiprocessor systems. In *35th Annual Simulation Symposium*, Apr. 2002.

[80] A. J. Oliner, L. Rudolph, and R. K. Sahoo. Cooperative checkpointing: A robust approach to large-scale systems reliability. In *International Conference on Supercomputing*, June 2006.

[81] K. Öner, L. Barroso, S. Iman, J. Jeong, K. Ramamurthy, and M. Dubois. The design of RPM: an FPGA-based multiprocessor emulator. In *International Symposium on Field Programmable Gate Arrays*, Feb. 1995.

[82] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *International Symposium on Computer Architecture*, Jun 2000.

[83] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, 1988.

[84] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with cache memories. In *International Symposium on Computer Architecture*, June 1984.

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

[86] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing*, Nov. 2003.

[87] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale. NAMD: biomolecular simulation on thousands of processors. In *Supercomputing*, Nov. 2002.

[88] A. Pimentel and L. Hertzberger. Distributed simulation of multicomputer architectures with mermaid. In *Symposium on Performance Evaluation of Computer and Telecommunication Systems*, July 1998.

[89] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *International Symposium on Microarchitecture*, Dec. 2001.

[90] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *ACM SIGMETRICS*, May 1993.

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

[92] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.

[93] A. D. Samples. Mache: no-loss trace compaction. *ACM SIGMETRICS Performance Evaluation Review*, 17(1):89–97, 1989.

[94] K. Sayood. *Introduction to data compression*. Morgan Kaufman Publishers, second edition, 2002.

[95] J. Seward. bzip2 1.0.2. `http://www.bzip.org`, 2001.

[96] A. Seznec. Analysis of the o-geometric history length branch predictor. *ACM SIGARCH Computer Architecture News*, 33(2):394–405, 2005.

[97] T. Sherwood and B. Calder. Loop termination prediction. In *International Symposium on High Performance Computing*, Oct. 2000.

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

[99] D. Shkarin. PPM: one step to practicality. In *Data Compression Conference*, 2002.

[100] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, 1999.

[101] K. Skadron, M. Martonosi, and D. W. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-Level Parallelism*, 2, 2000.

[102] D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. In *International Symposium on Workload Characterization*, 2005.

[103] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[104] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, 1991.

[105] R. Srinivasan, J. Cook, and S. Cooper. Fast, accurate microarchitecture simulation using statistical phase detection. In *International Symposium on Performance Analysis of Systems and Software*, Mar 2005.

[106] Standard Performance Evaluation Corporation. CPU2000, Dec. 1999.

[107] J. W. Stark. personal communication via email, Oct. 2005.

[108] J. W. Stark and C. Wilkerson et al. The 1st JILP championship branch prediction competition. In *Workshop at MICRO-37 and in Journal of Instruction Level Parallelism*, Jan. 2005. `http://www.jilp.org/cbp/`.

[109] O. O. Sudakov and E. S. Meshcheryakov. CHPOX: CHeckPOinting for linuX. `http://www.cluster.kiev.ua/tasks/chpx_eng.html`, Dec. 2004.

[110] R. A. Sugumar. *Multi-Configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs*. PhD thesis, University of Michigan, Aug. 1993. Technical Report CSE-TR-173-93.

[111] P. Sweazey and A. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *International Symposium on Computer Architecture*, June 1986.

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

[113] The BlueGene/L Team. An overview of the IBM BlueGene/L supercomputer. In *Supercomputing*, Nov. 2002.

[114] J. G. Thompson. *Efficient analysis of caching systems*. PhD thesis, University of California at Berkeley, 1987.

[115] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *International Conference on Supercomputing*, 2005.

[116] Y. Tse-Yu and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *International Symposium on Computer Architecture*, May 1992.

[117] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A pre-silicon software development environment for the IA-64 architecture. *Intel Technology Journal*, 3(4), 1999.

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

[119] M. Van 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.

[120] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *International Symposium on Performance Analysis of Systems and Software*, Mar. 2006.

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

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

[123] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

[124] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical sampling of computer system simulation. *To appear: IEEE Micro*, 26(4), July/August 2006.

[125] T. W. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *International Symposium on Performance Analysis of Systems and Software*, 2006.

[126] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.

[127] R. Wunderlich et al. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture*, June 2003.

[128] M. T. Yourst. *PTLsim User's Guide and Reference*, 2006. `http://ptlsim.org`.

[129] M. T. Yourst. *PTLsim/X: Xen and the Art of Full System Multiprocessor Simulation*, 2006. `http://ptlsim.org/PTLsimXen.pdf`.

[130] M. Zhang. *Latency Reduction Techniques for Single-Chip Multiprocessor Cache Systems*. PhD thesis, Massachusetts Institute of Technology, Dec. 2005.

[131] X. Zhang and R. Gupta. Whole execution traces. In *International Symposium on Microarchitecture*, Dec. 2004.

[132] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *International Parallel and Distributed Processing Symposium*, April 2004.