STATISTICAL SAMPLING OF MICROARCHITECTURE PERFORMANCE SIMULATION

ROLAND E. WUNDERLICH

MAY 2010

..

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

CARNEGIE MELLON UNIVERSITY

CARNEGIE INSTITUTE OF TECHNOLOGY

ELECTRICAL AND COMPUTER ENGINEERING

Abstract

Software-based microarchitecture performance simulators are many orders of magnitude slower than the hardware they simulate. Hence, most microarchitecture design studies draw their conclusions from simulations of truncated benchmark applications or ad-hoc instruction stream subsets that produce misleading or inaccurate results. We developed a statistical sampling based framework for microarchitecture simulation to enable fast, accurate, and reliable performance (cycles and energy per instruction) measurements of full-length benchmarks. We found that large samples of random brief measurements is the most effective sample design allowing measurement of far less than 0.1% of benchmark application instructions while achieving 99.7% confidence of ±3% error. We investigated several bias-free warming methodologies, and developed an experimental methodology to empirically determine warming needs. Finally, we evaluated the effectiveness of phase detection and stratified sampling at reducing sample size while maintaining accuracy.

Acknowledgements

I would like to thank my committee for their assistance with this thesis: James Hoe, Babak Falsafi, Markus Püschel, and David Brooks. I have benefitted significantly from the teaching, mentorship, and most of all, the example of my advisor James Hoe.

The SMARTS work was done in close collaboration with fellow graduate student, Thomas Wenisch, who was advised by Babak Falsafi.. Funding for this work was provided by an Intel Corporation PhD Fellowship.

My collaborations with and the support of my fellow students at Carnegie Mellon were especially valuable. Thank you Chi Chen, Shelley Chen, Eric Chung, Mike Ferdman, Chris Gniady, Brian Gold, Nikos Hardavellas, Jangwoo Kim, Peter Milder, Eriko Nurvitadhi, Joy-deep Ray, Jared Smolens, Stephen Somogyi, Tom Wenisch, and Se-Hyun Yang.

Contents

List of tables

List of figures

1 Introduction

Computer architects have long relied on software simulation to study the functionality and performance of proposed microarchitecture designs. Despite phenomenal improvement in processor performance over the last decades, the disproportionate growth in hardware complexity that needs to be modeled, most specifically parallelism, has steadily eroded simulation speed. Today, the fastest cycle-accurate uniprocessor performance simulators are more than five orders of magnitude slower than the hardware they model—requiring thousands of executed instructions per simulated instruction. Full system simulators and register-transfer-level simulators are easily six or more orders of magnitude slower than the proposed hardware. One minute of execution in real time can correspond to days, if not weeks, of simulation time.

Simulators of larger, and more parallel, processors such as systems on a chip and multi-core processors have even greater performance issues. The obvious optimization of using multiple threads in the software simulator to take advantage of parallel execution resources is very difficult due to the constant cycle level communication between chip resources. Thus, it is not anticipated that simulators will trend towards a smaller performance gap with the hardware they model.

We decided to view the performance of processors running benchmark applications as a large data set that we can statistically sample to more rapidly estimate overall performance. We have analyzed the statistical properties of both time and energy performance of super-scalar out-of-order microprocessors to determine the optimal random sampling and stratified sampling strategies.

We discovered that a large sample size (e.g., 10,000) of short performance measurements over the full length of a benchmark minimized the number of instruction executions that need to be measured. However, the accurate warming of microarchitectural state before each of these measurements was challenging. The warming overhead requirements for each measurement reduce the optimal sample size when optimizing simulation latency.

1.1 Current approaches

To mitigate prohibitively slow simulation speeds, researchers often use abbreviated or ad-hoc subsets of instruction execution streams of benchmarks as representative workloads in design studies. Unfortunately, many studies [Lauterbach 1994; Conte, Hirsch, and Menezes 1996; Lafage and Seznec 2000; Sherwood et al. 2002] have repeatedly concluded that results based only on a single abbreviated execution stream are inaccurate or misleading because they fail to capture global variations in program behavior and performance.

Another common approach to curtail simulation time is to use fewer or smaller input sets (i.e., the test or train sets rather than all of the reference sets in SPEC CPU2000). However, research has shown benchmark behavior varies significantly across test, train and reference inputs for a number of SPEC CPU2000 benchmarks [Hsu, Chen, and Yew 2002; Sherwood et al. 2002].

To obtain performance results based on complete benchmarks and input sets, many proposals have advocated statistical [Laha, Patel, and Iyer 1988; Lauterbach 1994; Conte, Hirsch, and Menezes 1996; Haskins and Skadron 2001] or profile-driven [Lafage and Seznec 2000; Hamerly et al. 2005; Falcón, Faraboschi, and Ortega 2007] simulation sampling. Simulation sampling measures only chosen sections (called sampling units) from a benchmark's full execution stream. The sections in between sampling units are "fast-forwarded"

using functional simulation that only maintains programmer-visible architectural state. We faced two key challenges to simulation sampling: (1) choosing an appropriate subset with the minimum number of instructions to meet a given error bound, and (2) reconstructing an accurate microarchitectural state (e.g., branch predictor and cache hierarchy contents) for unbiased sample measurement following an extended period of functional fast-forwarding.

Existing proposals for simulation sampling suffer from several key shortcomings. On the efficiency front, most proposals sample several orders of magnitude more instructions than are statistically necessary for their stated error [Laha, Patel, and Iyer 1988; Lauterbach 1994; Lafage and Seznec 2000; Haskins and Skadron 2001; Hamerly et al. 2005]. This inefficiency is often rooted in their excessively large sampling units, either to amortize the overhead of reconstructing microarchitectural state or to capture coarse-grain performance variations by brute force. On the accuracy front, most proposals either do not offer tight error bounds on their performance estimations [Laha, Patel, and Iyer 1988; Lauterbach 1994; Lafage and Seznec 2000; Hamerly et al. 2005; Falcón, Faraboschi, and Ortega 2007], or require unrealistic assumptions about the microarchitecture (e.g., perfect branch prediction or cache hierarchies) [Conte, Hirsch, and Menezes 1996].

1.2 The SMARTS approach

We propose the *Sampling Microarchitecture Simulation* (SMARTS) framework which applies statistical sampling theory to address the aforementioned issues in simulation sampling. Unlike prior approaches to simulation sampling, SMARTS prescribes an exact and constructive procedure for selecting a minimal subset from a benchmark's instruction execution stream to achieve a desired confidence interval. SMARTS uses a measure of variability (coefficient of variation) to determine the optimal sample that captures a program's inher-

ent variation. An optimal sample generally consists of a large number of small sampling units. Unbiased measurement of sampling units as small as 1000 instructions is possible by applying careful *functional warming*—maintaining large microarchitectural state, such as branch predictors and the cache hierarchy—during fast-forwarding between sampling units.

We evaluate SMARTS in the context of a wide-issue superscalar out-of-order timing simulator called SMARTSim. We employed SMARTSim to estimate the CPI and energy per instruction (EPI) for 41 out of 45 SPEC CPU2000 benchmark/input combinations [Henning 2000] on two microarchitecture configurations. We make the following primary observations:

1. **Optimal sampling:** SMARTSim achieves an actual average error of only 0.64% on CPI and 0.59% on EPI by simulating fewer than 50 million instructions in detail for each of the 41 SPEC CPU2000 benchmarks. This represents an exceedingly small fraction of the complete benchmark streams, which are 174 billion instructions on average (Alpha ISA).
2. **Simulation speedup:** SMARTSim can achieve average speeds of 35 and 60 times faster relative to sim-outorder for 8-way and 16-way superscalar processor models, respectively.
3. **Implications:** SMARTS sampling simulation rate is, for all practical purposes, decoupled from the speed of the detailed simulator. This result has fundamental bearings on future simulator designs. First, designers should focus less on elaborate performance shortcuts in detailed simulators, and more on increasing the detailed simulator's overall design flexibility and accuracy. Second, designers should focus on developing techniques which speed up fast-forwarding and functional warming

(e.g., native execution [Reinhardt et al. 1993; Chen 2004; Falcón, Faraboschi, and Ortega 2007], simulator synthesis [Burtscher and Ganusov 2004], and checkpointing [Van Biesbrouck, Eeckhout, and Calder 2005] (Chapter 4), as these ultimately determine sampling simulation rate.

1.3 Checkpoint sampling

Although functional warming enables accurate performance estimation, it limits SMARTS' speed, occupying more than 99% of simulation runtime. Functional warming dominates simulation time because SMARTS must functionally simulate the entire benchmark's execution, even though it will simulate only a tiny fraction of the execution using detailed microarchitecture timing models.

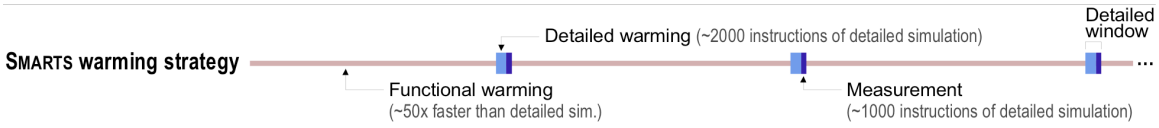


Figure 1.1: SMARTS two-tier warming strategy

Functional warming dominates runtime since it must cover billions of instructions per benchmark application.

The second shortcoming of the SMARTS framework is that functional warming requires simulation time proportional to benchmark length rather than sample size. As a result, the overall runtime of a SMARTS experiment remains constant even when we reduce the measured sample size—for example, by relaxing an experiment's statistical confidence requirements. Moreover, functional warming time increases with the adoption of benchmark suites, such as SPEC CPU2006, that lengthen benchmarks to scale with hardware performance improvement.

We developed our own version of checkpoints, *live points*, to enable rapid loading of warm microarchitecture state before each measurement. This approach also enables

random measurement ordering and parallel simulation. Live points provide an alternative to functional warming that reduces simulation turnaround time without sacrificing accuracy. A live point stores the necessary data to reconstruct warm state for a simulation sampling execution window. Although modern computer architecture simulators frequently provide checkpoint creation and loading capabilities, existing checkpoint implementations have two limitations:

1. They don't provide complete microarchitectural model state.
2. They cannot scale to the required checkpoint library size (about 10,000 checkpoints per benchmark), which would require multiple terabytes of storage.

We address the first limitation by storing only selected microarchitectural state in live points, an approach we call *checkpointed warming*. The key challenge of checkpointed warming lies in storing microarchitectural state such that live points can still simulate the range of microarchitectural configurations of interest. Fortunately, with the exception of the branch predictor and memory hierarchy, most microarchitectural state can be reconstructed dynamically with minimal simulation (a few thousand instructions of detailed warming), and thus need not be stored. For the exceptional structures, researchers can often place limits on the configurations of interest (for example, through trace-based studies). We've designed checkpointed warming to reproduce these structures under user-specified limits.

We reduce the size of conventional checkpoints by three orders of magnitude through storing in live points only the subset of state necessary for limited execution windows, an approach we call *live state*. Live state exploits the brevity of simulation sampling execution windows (thousands of instructions) to omit most state.

1.4 Stratified sampling

Simple random sampling does not exploit the often repetitive behaviors of benchmarks, collecting many redundant measurements. By identifying repetitive behaviors, we can apply stratified random sampling to achieve the same confidence as simple random sampling with far fewer measurements.

Our oracle limit study of optimal stratified sampling of SPEC CPU2000 benchmarks demonstrates an opportunity to reduce required measurement by 43 times over simple random sampling. Using our oracle results as a basis for comparison, we evaluate three practical approaches for selecting strata, offline online and program phase detection, and IPC profiling.

Offline phase detection involves profiling a benchmark application to determine microarchitecture independent phases in the instruction stream. We evaluated the SimPoint approach [Sherman et al. 2002] that identifies phases with similar groups of dynamic instruction basic blocks and compared it's confidence intervals versus simple random sampling. We find that approaches based on microarchitecture dependant features for phase detection will fundamentally have a large potential for error and only have limited improvement in sample size due to a lack of phase variance minimization.

Online phase detection performs the phase identification simultaneously with the performance simulation itself. We evaluated the dynamic sampling approach [Falcón, Faraboschi, and Ortega 2007] that identifies measurement periods by monitoring the an instruction translation cache miss rate. Dynamic sampling also does not provide an obvious way to estimate confidence intervals.

Finally, we developed a framework called IPC profiling that directly minimizes stratum variance, therefore minimizing sample size. Our results indicate that: (1) program phase stratification falls short of optimal opportunity, (2) IPC profiling requires expensive microarchitecture-specific analysis, and (3) all three methods require large sampling unit sizes to make strata selection feasible, offsetting their reductions of sample size. We conclude that, without better stratification approaches, stratified sampling does not provide a clear advantage over simple random sampling.

1.5 High-resolution tracing

Simulation sampling provides high level aggregate performance estimates that are most effective when comparing microarchitecture or software changes. We present a digression in the opposite direction, our efforts on novel approaches to capturing detailed performance data in-band on real computer processors, not simulations, to aid in the tuning of software applications and microarchitectures.

We investigated the fidelity of hardware performance counter data on the Intel Core 2 processor architecture. At resolutions smaller than 100,000 instruction, the signal to noise ratio (SNR) is over 10dB, producing unreliable and unusable data for further analysis. We developed two reconstruction approaches, median reconstruction and super-resolution reconstruction, that can improve the SNR and resolution of performance counter trace data. We present our preliminary results with these techniques.

1.6 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 presents background on microarchitecture simulation and statistical sampling theory. Chapter 3 presents our initial

studies of uni-processor simulation sampling (SMARTS) using two-phase warming that we call functional warming. Chapter 4 presents our investigation of the advantages and best design of checkpoints for sampling, including checkpointed warming, and parallelizable sample design. Chapter 5 presents our most recent work on stratified sampling approaches that reduce sample size. Finally, we present our tangent work on high-resolution performance tracing in Chapter 6. Our conclusions are presented in Chapter 7.

2 Background

2.1 Microarchitecture simulation

Designing the microarchitectures of modern processors requires the evaluation of many complex engineering decisions that calls for large research efforts. Assessing design trade-offs and architectural options is a multi-year process for each new project. This evaluation effort spans many levels of abstraction between concept and implementation. Academic and industrial design efforts tackle the complexity of evaluating new CPU architectures by progressively reducing the level of abstraction being studied. Starting with simple architectural concepts, new ideas are evaluated through simulation and implementations at various levels of detail. The accuracy of performance metrics is improved by increasing the level of detail being modeled. The more detailed models often require large leaps in complexity or description size, and thus, require a large effort to produce. Hence, in many cases early design decisions, ones that influence the broad direction of a project, are made with only loose estimates of performance.

Software simulation is the most commonly used tool in early microarchitecture evaluation. Simulations are extremely flexible, and can be developed and modified with only modest effort. In addition, they offer great flexibility in the level of detail they can model. The two most common types of software simulation are those that are functionally accurate, and those that are timing accurate. Functional simulators implement the semantics of the target instruction set architecture. Timing simulators attempt to estimate the performance of microarchitectures with varying levels of detail and accuracy.

The parallel nature of simulating processor logic in software is not a good fit for the sequential execution of the simulator's instruction stream. Coupled with the thousands of instructions necessary to simulate the performance characteristics of each simulated instruction, the reasons for the speed gap between simulators and processors is evident. Full system simulators which encompass the entire computer system, and not just the processor, encounter even larger slowdowns.

Two often used approaches to speeding up simulation are checkpointing and native execution. Checkpointing stores all needed simulator state on disk for later use to rapidly resume simulation deep within a benchmark application, instead of always starting at the beginning of a benchmark (Chapter 4). This enables simulation of more interesting portions of a benchmark with lower latency.

Native execution, as well as just-in-time translation or compiling, is most effective for functional simulation [Chen 2004]. Native execution involves performing the semantics of the "simulated" instructions directly on an underlying processor. Therefore, the direct execution requires ISA compatibility between the underlying computer system with the simulated ISA. Native execution is also largely incompatible with producing timing accurate results since the means of instrumenting the "simulated" instructions execution is with inline instrumentation or traps. These instrumentations are essentially a non-native execution simulator, thus amortizing any speed advantage of native execution.

2.2 Statistical sampling

The field of inferential statistics offers well-defined procedures to quantify and to ensure the quality of sample-derived estimates. This section provides basic background on statisti-

cal sampling. We describe procedures for selecting a sample for mean estimation and the mathematics for calculating the confidence in an estimate.

Statistical sampling attempts to estimate a given cumulative property of a *population* by measuring only a *sample*, a subset of the population [Jain 2001]. By examining an appropriately selected sample, one can infer the nature of the property over the whole population in terms of total, mean, and proportion. The theory of sampling is concerned with choosing a minimal but representative sample to achieve a quantifiable accuracy and precision in the estimate. The theory does not presume a normally-distributed population. Our goal is to apply this theory to: (1) identify a minimal but representative sample from the population for microarchitecture simulation, and (2) establish a confidence level for the error on sample estimates.

Table 2.1: Statistical sampling terminology and variables

Sampling terminology

population	complete set of elements with a property to be estimated
sample	measured subset of a population
element / sampling unit	quantum of the population that is measured in a sample

Sample types

simple random sampling	sampling units chosen at random from whole population
systematic sampling	sampling units chosen at a periodic interval
uniform sampling	sampling units chosen with equal probability from entire pop.
representative sampling	sampling units chosen from weighted regions of the pop.
stratified sampling	sampling units chosen from strata [Wunderlich et al. 2004]

Population variables

N	population size
$\overline{X}$	mean
σ_x	standard deviation
V_x	coeff. of variation ($\sigma_x / \overline{X}$)

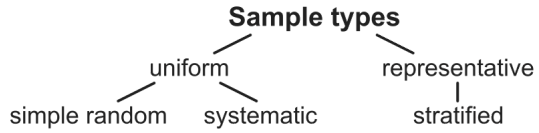
Useful relations

$$\pm\varepsilon \cdot \overline{X} = \pm \frac{z \cdot \hat{V}_x \cdot \overline{x}}{\sqrt{n}}$$

confidence interval for a sample

$$n \geqslant \left(\frac{z \cdot \hat{V}_x}{\varepsilon} \right)^2$$

sample size for a subsequent experiment



Sample variables

n	sample size
$\overline{x}$	sample mean
$\hat{V}_x$	sample coeff. of variation ($\hat{\sigma}_x / \overline{X}$)
$(1 - \alpha)$	confidence level (95% z=1.97, 99.7% z=3)
$\pm\varepsilon \cdot \overline{X}$	confidence interval
j, k	systematic sampling offset, interval
$B(\overline{x})$	bias of sample mean

Table 2.1 summarizes the standard statistical sampling terminology and variables relevant to this study. *Simple random sampling* selects a sample of n elements (a.k.a. sampling units) at random from a population of N elements. Measurements are taken on the selected sampling units, and for a sufficiently large sample size (i.e., $n > 30$) the sampled results can be meaningfully extrapolated to provide an estimate for the whole population. In particular, the true population mean $\overline{X}$ of a property X is estimated by the sample mean $\bar{x}$. The coefficient of variation is the standard deviation of X normalized by $\overline{X}$, $V_x = \sigma_x / \overline{X}$. The likelihood that $\bar{x}$ is a good estimate of $\overline{X}$ improves with sample size and decreases with V_x. SMARTS leverages the relationship between n, V_x and desired confidence to minimize the required sample size for a benchmark.

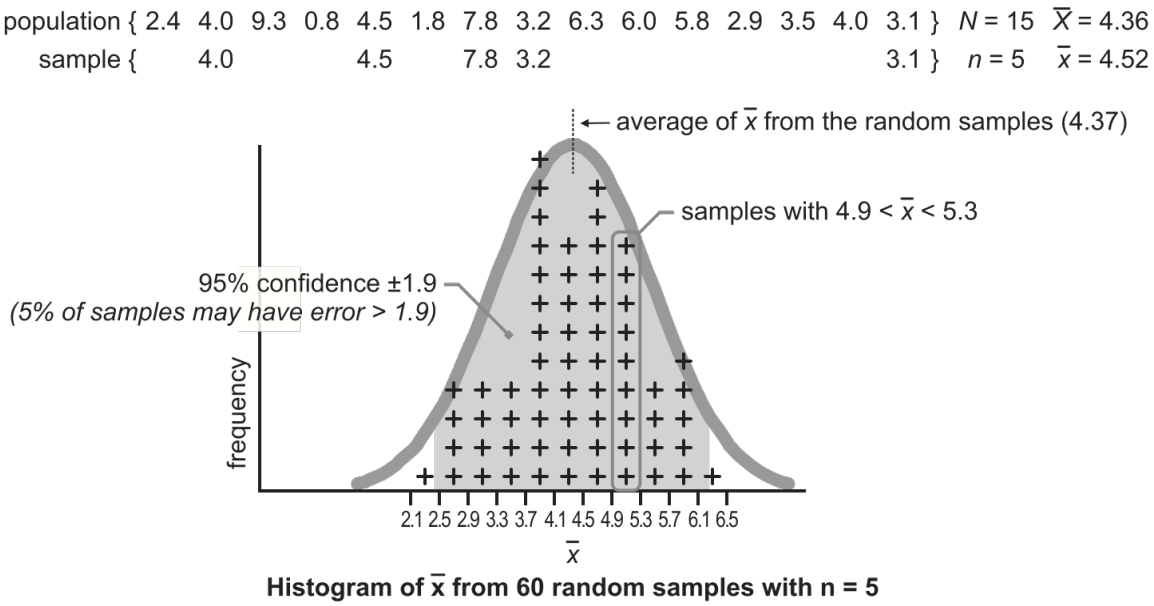


Figure 2.1: Confidence interval for uniform sampling error

The confidence interval represents the probability of a sample's estimate being within a specified range of the actual population's property value.

Formally, the confidence in a mean estimate is jointly quantified by two interdependent terms: confidence level $(1 - \alpha)$ and confidence interval $\pm\varepsilon \cdot \bar{X}$. The interpretation of confidence level and interval is that, over a large number of random sampling trials, a $(1 - \alpha)$ fraction of the trials should produce $\bar{x}$ that is within $\pm\varepsilon \cdot \bar{X}$ of $\bar{X}$.[1] Figure 2.1 graphically illustrates obtaining an estimate of $\bar{X}$ with a sample, and an interpretation of the confidence interval. The confidence interval achieved by a sample is $\pm([(z \cdot V_x)/\sqrt{n}] \cdot \bar{X})$ where z is the $100[1 - (\alpha / 2)]$ percentile of the standard normal distribution. (We assume $N \gg n \gg 1$ to simplify the expressions in this study.) For a sample with a given V_x and size n, one can choose a desired confidence level and solve for the achieved confidence interval.

[1] A less rigorous, but acceptable interpretation is that for a given sample there is a $(1 - \alpha)$ probability that $\bar{x}$ is within $\pm\varepsilon \cdot \bar{X}$ of $\bar{X}$.

To design a sampling simulation to meet a certain confidence, one begins by determining an appropriate *n* based on the required confidence and V_x, using the same equations above. (Note that the population size does not impact the determination of *n*.) The true coefficient of variation of a population is rarely available in practice unless the entire population is examined. Instead, $\hat{V}_x$ of a sufficiently large initial sample is commonly used in place of V_x in computing the confidence of that sample. If the initial sample does *not* achieve the desired confidence, the required size of a subsequent sample can be computed using $\hat{V}_x$, where $n \geq ((z \cdot \hat{V}_x)/\varepsilon)^2$. In practice, the required sample size can typically be found after one test sample.

An approximation of random sampling of practical interest in microarchitecture simulation is *systematic sampling* due to its ease of implementation. This approach selects sampling units from an ordered population at a fixed sampling interval *k* such that $n = N/k$. Systematic sampling is most effective if the population exhibits low homogeneity. In other words, the measured property *X* should not vary cyclically over the population sequence at the same periodicity as *k* or its higher harmonics. Homogeneity in a population is quantified by the intraclass correlation coefficient δ_x; when the magnitude of δ_x is negligible, the confidence calculations for systematic sampling are the same as described for random sampling. We verified experimentally that in our sampling results the population exhibits negligible homogeneity on the order of -1×10^{-6}. This observation agrees with our intuition that realistic benchmarks do not have sufficiently regular cyclic behavior at the periodicity relevant to simulation sampling (tens of millions of instructions).

Measurement error is another source of inaccuracy for both random and systematic sampling. Random errors lead to an increase in $\hat{V}_x$ and are accounted for by a correspondingly lowered confidence in the estimate. On the other hand, systematic errors—for

example, due to incorrect cache hierarchy state prior to the start of a sampling unit [Laha, Patel, and Iyer 1988]—introduce a bias in the estimate. The bias $B(\bar{x})$ is the average difference between $\overline{X}$ and $\bar{x}$ over all possible sampling trials of a given configuration. For systematic sampling, there are exactly k possible systematic sample phases, and hence, $B(\bar{x}) = \sum \bar{x}/k - \overline{X}$. If bias is known, it can be accounted for by subtracting it from the estimate, without affecting confidence. If the bias can only be bounded, then it introduces a proportional amount of uncertainty in the estimate beyond the confidence interval.

2.3 Sampling approaches

A rigorous approach to obtaining representative estimates when sampling was performed by Conte, Hirsch, and Menezes [1996]. In addition, their study used execution-driven microarchitecture simulation as opposed to trace-based simulation. Conte's sample design addressed both sampling error from insufficient sample size, and non-sampling bias due to unwarmed state at the start of measurements. Sampling error was effectively brought to reasonable levels by taking ~1000 measurements of at least 2000 instructions each while simulating SPEC CPU95. SMARTS extends the sampling parameter search across a much larger range of possible sample sizes and sampling unit sizes to determine the optimal values for SPEC CPU2000 with a more complex and modern microarchitecture simulator. The Conte study does not address cold-start bias for caches, and assumes a perfect memory hierarchy. However, Conte found that a two-level branch predictor was effectively warmed by 7000 instructions of detailed simulation before each measurement, a warming technique similar to Smart's detailed warming.

All the works cited in this section, Section 3.4, and SMARTS, use a uniform sampling approach where measurements are taken randomly or systematically from the whole

instruction stream. However, it is possible to reduce the required amount of measurement if low-variance phases can be identified [Wunderlich et al. 2004]. Alternatively, practical constraints sometimes make the simulation of 1000's of measurements undesirable in comparison to fewer carefully-selected measurements. For example, it may be difficult to extract 1000's of checkpoints collected by scanning architectural and microarchitectural state from a physical processor. Instead, careful profiling may identify performance-critical program phases, allowing only a few checkpoints corresponding to the selected phases. The use of individual measurements of program phases may also be appropriate when the goal is simply to simulate a design on various types of dominant program behavior, and representative benchmark performance is not required.

Two significant works in identifying program phases for representative sampling are [Iyengar, Trevillyan, and Bose 1996] and [Hamerly et al. 2005]. Iyengar, Trevillyan, and Bose [1996] developed a composite metric, the R-metric, to measure how representative trace-subsets are as compared to the whole instruction stream. The R-metric compared the basic-block occurrence frequencies between each subset versus the whole program. Iyengar developed a graph-based selection algorithm for subsets with optimal R-metric values. Hamerly et al. [2005], describe a clustering-based algorithm to identify instruction stream regions that have similar basic block occurrence frequencies. Hamerly et al. selected measurement locations after identifying similar program regions by clustering basic-block relative frequency vectors. Both of these approaches cannot achieve the high level of accuracy and reliability of statistical sampling, but are advantageous when collecting many measurements is infeasible.

Both [Iyengar, Trevillyan, and Bose 1996] and [Hamerly et al. 2005] rely on a high correlation of performance to repeated program instructions to achieve accurate performance estimates. The program phases they identify are composed of program regions

estimates. The program phases they identify are composed of program regions with similar basic-block occurrence frequencies. If these phases do not contain homogeneous performance then small samples will not produce accurate estimates, as seen in Section 3.3.3. However, it has been shown that in most cases there is a strong correlation between BBV-identified phases and performance [Lau et al. 2005]. A survey of prevailing simulation-sampling approaches by Yi et al. [2005] concluded that the SMARTS simulation-sampling approach provides the highest estimation accuracy.

3 SMARTS sampling

This chapter describes our investigation of simple random sampling of uniprocessor microarchitecture performance simulations with an aim to minimizing the number of measured instructions.

3.1 Framework

This section presents a framework for Sampling Microarchitecture Simulation (SMARTS). SMARTS applies statistical sampling to accelerate simulation-based performance measurements. Our presentation of SMARTS is primarily developed around estimating average cycles per instruction (CPI), but we provide results in Section 3.3.2 for estimating both CPI and energy per instruction (EPI). The SMARTS framework is generally applicable to other performance metrics, such as pipeline resource utilization or average memory latency.

3.1.1 Technique

Measuring the CPI of a benchmark's full instruction stream on a detailed microarchitecture simulator is a time-consuming proposition. SMARTS estimates the CPI in significantly less time by simulating and measuring only a tiny fraction of the stream on the detailed microarchitecture simulator. SMARTS assumes an execution-driven simulator that supports *detailed* simulation and *functional* simulation (a.k.a. fast-forwarding). In the detailed mode all relevant microarchitecture details are accounted for. Only programmer-visible architectural state (e.g., architectural registers and memory) is updated in the functional mode. SMARTS uses the two simulation modes to sample CPI systematically at a fixed interval—

detailed simulation of the sampled instructions and functional simulation of the remaining instructions.

Table 3.1: Variables introduced or redefined in the SMARTS framework

U	sampling unit size (instructions)
W	detailed warming (instructions)
N	benchmark length (instructions) / U

SMARTS uses systematic sampling rather than random sampling because systematic sampling is more straight-forward to implement in execution-driven simulators. In SMARTS, a sampling unit is defined as U consecutive instructions in a benchmark's dynamic instruction stream such that the population size N is the length of the stream divided by U. The exact number of instructions per sampling unit may vary slightly to align sampling units on clock cycle boundaries. For systematic sampling at an interval k, beginning at offset j, SMARTS repeatedly alternates between a functional simulation period of $U(k-1)$ instructions and a detailed simulation/measurement period of U instructions. A primary reason we base the population on instructions rather than clock cycles is that one cannot meaningfully count the number of detailed cycles elapsed during functional simulation.

Evaluating benchmarks in SMARTS provides an estimated average CPI based on the $n \cdot U$ sampled instructions. Equally important, the results include the measured coefficient of variation $\hat{V}_{CPI}$, that allows us to calculate the confidence of the CPI estimate, and if necessary, determine a new sample size to meet a specific degree of confidence. Section 3.3.1 describes how to set SMARTS sampling parameters and prescribes an exact procedure to generate an accurate performance estimate by measuring only a minimal subset of a benchmark's instruction stream.

A key challenge in SMARTS is how to compute the correct microarchitectural state prior to detailed measurement of each sampling unit. Between sampling units, functional simulation computes all architectural state updates of the program, but leaves microarchitectural state (e.g., cache hierarchy, branch predictors and target buffers, or pipeline state) unchanged. Stale microarchitectural state introduces a large bias in the measurement of individual sampling units and, consequently, the final estimate. We have observed stale-state induced bias as high as 50% for sampling units of 10,000 instructions.

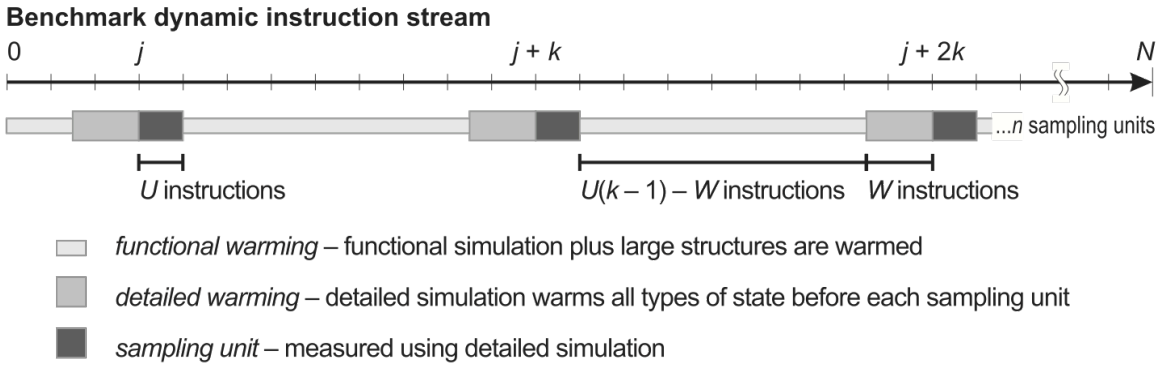


Figure 3.1: Systematic sampling as performed in SMARTS

Two modes of simulation are used: functional simulation, and detailed simulation. The need to determine warming requirements for large structures, such as caches, is eliminated by performing continuous functional warming.

The stale-state effect can be ameliorated by introducing a warming period where W instructions are simulated in detail to refresh the microarchitectural state just prior to the measurement of a sampling unit [Laha, Patel, and Iyer 1988]. We refer to this solution as detailed warming. Figure 3.1 graphically illustrates how SMARTS alternates between functional simulation of $[U(k-1)-W]$ instructions, detailed simulation of W warming instructions (without measurement), and detailed simulation and measurement of U instructions. Increasing W can gradually reduce the bias below an acceptable threshold.

Unfortunately, detailed warming has two major shortcomings: (1) detailed warming can be expensive because it increases the amount of detailed simulation, and (2) in general the

appropriate value of *W* is difficult to derive analytically because some microarchitectural state has extremely long history. We will return to this discussion in Section 3.2.3, where we measure the effect of *W* on bias in a reference implementation of SMARTS.

Between detailed simulation periods, select microarchitectural state could instead be maintained by functional simulation with only a small overhead. We refer to this warming approach as *functional warming*. The cache hierarchies and branch predictors are prime candidates for functional warming. By continuously warming microarchitectural state with very long history, we can analytically bound *W* for the remaining state to a manageably small value.

A caveat to the functional warming approach is that it may not always be able to accurately reproduce the correct microarchitectural state if correct warming requires exact knowledge of detailed execution. Moreover, timing-dependant behavior (e.g., operating system scheduling activity) require timer approximation. If functional warming simulates instructions in order, it also may not accurately reflect the artifacts of out-of-order and speculative event ordering. Cain et al. [2002] have suggested that out-of-order and speculative ordering has minimal impact on CPI and other performance metrics. In Section 3.2.5 we corroborate these results and present our own analysis of the residual biases after functional warming. We believe functional warming is the most cost-effective approach to achieve accurate CPI estimation with simulation sampling.

3.1.2 Benchmarks

In this study, we demonstrate the effectiveness of SMARTS by attempting to estimate the CPI and EPI of the SPEC CPU2000 integer and floating-point benchmarks [Henning 2000] as measured on the SimpleScalar 3.0 sim-outorder simulator [Burger and Austin 1997] with

the Wattch 1.02 power estimation extensions [Brooks, Tiwari, and Martonosi 2000]. For improved realism, we modified the memory subsystem to include a store buffer and miss status holding registers (MSHR), and model interconnect bottlenecks in the memory hierarchy. Our study includes the cross product of two microarchitecture configurations and all 26 SPEC CPU2000 benchmarks as in Table 3.2. We evaluate all reference inputs except vpr-place and three perlbmk inputs, as these inputs fail to simulate correctly in sim-outorder. Overall, 41 benchmark/input set combinations are included in this study. To provide a reference data set for this study, we collect cycle-by-cycle traces of instruction commits in sim-outorder for the entire length of each benchmark. Simulating these SPEC CPU2000 benchmarks resulted in more than 7 trillion simulated instructions per machine configuration.

Table 3.2: Simulated SPEC CPU2000 benchmarks

Benchmark	Input	Instructions (bil.)	8-way IPC	16-way IPC	8-way EPI (nJ/Inst.)
ammp		326.5	1.04	1.60	42.7
applu		223.9	1.17	1.75	42.1
apsi		347.9	1.58	2.40	35.9
art-1	startx 110	41.8	0.39	0.66	88.0
art-2	startx 470	45.0	0.39	0.66	87.6
bzip2-1	source	108.9	1.48	1.77	39.5
bzip2-2	graphic	143.6	1.56	1.95	37.2
bzip2-3	program	124.9	1.70	2.08	36.9
crafty		191.9	2.34	2.94	34.6
eon-1	kajiya	101.3	2.50	3.11	36.5
eon-2	cook	80.6	3.01	4.81	31.8
eon-3	rushmeier	57.9	2.85	4.11	33.2
equake		131.5	0.79	1.23	50.3
facerec		211.0	1.86	3.31	33.2
fma3d		268.4	1.53	2.57	38.0
galgel		409.4	0.96	1.77	45.2
gap		269.0	1.48	1.74	39.5
gcc-1	166	46.9	1.45	1.41	40.7
gcc-2	200	108.6	1.60	1.82	39.6
gcc-3	expr	12.1	1.63	1.73	40.1
gcc-4	integrate	13.2	1.64	1.62	38.9
gcc-5	scilab	62.0	1.64	1.80	40.2
gzip-1	source	84.4	1.76	1.91	37.2
gzip-2	log	39.5	1.81	1.94	35.9
gzip-3	graphic	103.7	2.26	2.54	35.7
gzip-4	random	82.2	2.22	2.48	36.0
gzip-5	program	168.9	1.81	2.00	36.1
lucas		142.4	0.11	0.11	207.1
mcf		61.9	0.10	0.14	245.2
mesa		281.7	2.92	4.44	29.6
mgrid		419.2	1.75	2.91	36.3
parser		546.7	1.32	1.58	41.4
perlbmk	makerand	2.1	2.01	2.27	36.4
sixtrack		470.9	2.50	5.79	31.1
swim		225.8	1.03	1.51	42.8
twolf		346.5	0.76	0.83	52.7
vortex-1	lendian1	119.0	2.13	3.38	31.4
vortex-2	lendian2	138.7	2.35	3.89	30.7
vortex-3	lendian3	133.0	2.12	3.36	31.5
vpr	route	84.1	0.56	0.64	63.8
wupwise		349.6	2.37	4.26	30.4
mean		173.8			

The baseline microarchitecture configuration in this study is an 8-way superscalar model that represents a processor in the current technology generation. A 16-way superscalar configuration also is included to reflect an aggressive future design point. This configuration has a wider data path, larger out-of-order window, and larger caches, to test the effects of an enlarged state set. The details of the 8-way and 16-way configurations are summarized in Table 3.3.

Table 3.3: Simulated microarchitecture configurations

Parameter	8-way (baseline)	16-way
RUU/LSQ	128/64	256/128
Memory system	32KB 2-way L1I/D 2 ports 8 MSHR, 1M 4-way L2 16-entry store buffer	64KB 2-way L1I/D 4 ports 16 MSHR, 2M 8-way L2 32-entry store buffer
ITLB/DTLB	4-way 128 entries/ 4-way 256 entries 200 cycle miss	4-way 128 entries/ 4-way 256 entries 200 cycle miss
L1/L2/mem. latency	1/12/100 cycles	2/16/100 cycles
Functional units	4 I-ALU, 2 I-MUL/DIV 2 FP-ALU, 1 FP-MUL/DIV	16 I-ALU, 8 I-MUL/DIV 8 FP-ALU, 4 FP-MUL/DIV
Branch predictor	Combined 2K tables 7 cycle mispred. 1 prediction/cycle	Combined 8K tables 10 cycle mispred. 2 predictions/cycle

3.1.3 Speedup opportunity

The required sample size to estimate CPI at a given confidence is directly proportional to the square of the population's coefficient of variation, $n \propto V_{CPI}^2$. A benchmark with a small V_{CPI} implies a greater opportunity for accelerated simulation because fewer instructions from the benchmark need be simulated and measured in detail. To assess the potential speedup of SMARTS, we study V_{CPI} of all benchmarks in our test suite. A benchmark's instruction stream can be divided into a population using different values of U. Figure 3.2 plots V_{CPI} of all benchmarks on the 8-way configuration as a function of U in the range of 10 instructions to 1 billion instructions. V_{CPI} decreases with increasing U because short-term

CPI variations within a window of U instructions are hidden by averaging over the sampling unit. The V_{CPI} curves for all benchmarks share the same general shape, with a steep negative slope for U less than 1000, leveling off thereafter.

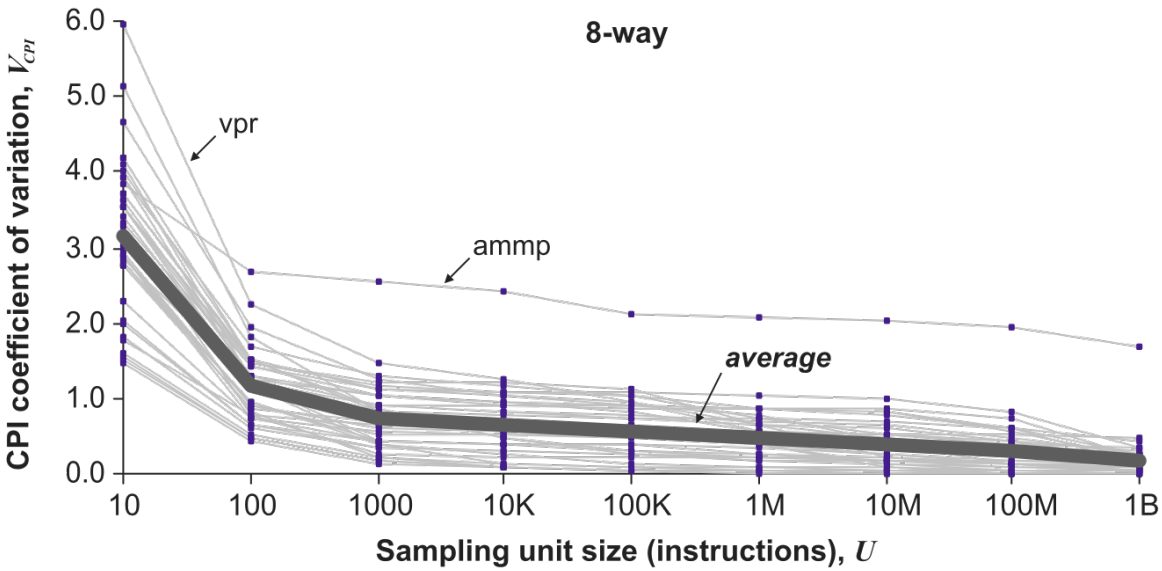


Figure 3.2: Coefficient of variation of CPI of SPEC CPU2000 benchmarks

Increasing the sampling unit size above 1000 instructions yields little reduction in V_{CPI}, and correspondingly small decreases in the required sample size. Therefore, simulation time usually increases for U > 1000 instructions.

The shapes of the V_{CPI} curves argue against sampling approaches that use large sample unit sizes because for U greater than 1000, V_{CPI} (and hence n) does not decrease rapidly enough to compensate for the increased sample unit size. For instance, although very few sampling units are required in the extreme case of $U = 1 \times 10^9$, the total number of sampled instructions $n \cdot U$ is much greater than when U is less than 1000. Figure 3.2 further makes the case that single-sampling-unit approaches, the most commonly employed approaches, cannot ensure accurate estimates since the coefficients of variation of many benchmarks are non-negligible even for sampling units of over one billion instructions.

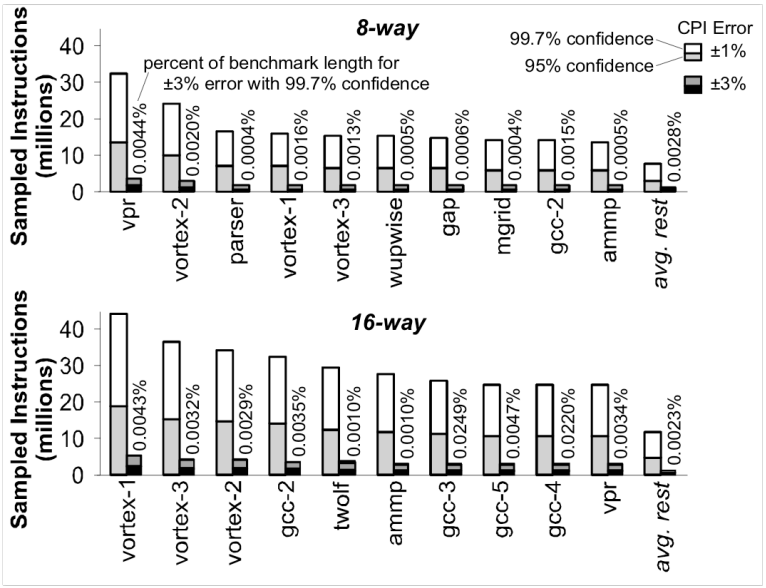


Figure 3.3: Minimum sampled instructions

The minimum number of instructions that must be measured to achieve commonly used confidence intervals, assuming no warming is needed for measurement, is an exceedingly small fraction of the SPEC CPU2000 benchmarks.

For $U = 10$, Figure 3.3 reports the values of $n \cdot U$ for all benchmarks, assuming several commonly used confidence targets. Even for a stringent confidence requirement of ±1% error with 99.7% confidence, the worst-case benchmark on the 8-way configuration in our study requires no more than 0.1% of its instruction stream to be measured. The number of instructions required to achieve a particular level of confidence does not vary significantly across benchmarks because, for the most part, the benchmarks have similar values of V_{CPI}. The exceedingly low detailed simulation requirement suggests that the simulation rate of SMARTS is insensitive to the speed of the detailed microarchitecture simulation. Rather, the rate depends on the speed of the functional simulation performed for the great majority of the instruction stream between sampling units. This optimistic assessment of speedup opportunity does not factor in the detailed simulation cost for microarchitectural state warming. We next present an analytical performance model for SMARTS to take into account the cost of detailed and functional warming.

3.1.4 Speedup model

We develop a SMARTS performance model to consider the trade-off presented by functional warming. Let $S_F \equiv 1.0$ represent the simulation rate of functional simulation, and S_D represent the simulation rate of detailed simulation relative to S_F. (Therefore, $1 / S_D$ is the slowdown of detailed simulation with respect to functional simulation.) The simulation rate of SMARTS using only detailed and no functional warming is given by $S_F([N - n(U+W)]/N) + S_D[(n(U+W))/N]$. This expression is a weighted average of S_F and S_D over the fraction of the instruction stream simulated functionally versus in detail. Figure 3.4 plots the SMARTS simulation rates for W between 0 and 10 million instructions for gcc-1, with $S_D = 1 / 60$ (corresponding to today's fastest detailed simulators) and $S_D = 1 / 600$ (projected simulation rate of future processor cores). The right-hand-side vertical axis estimates the corresponding runtimes.

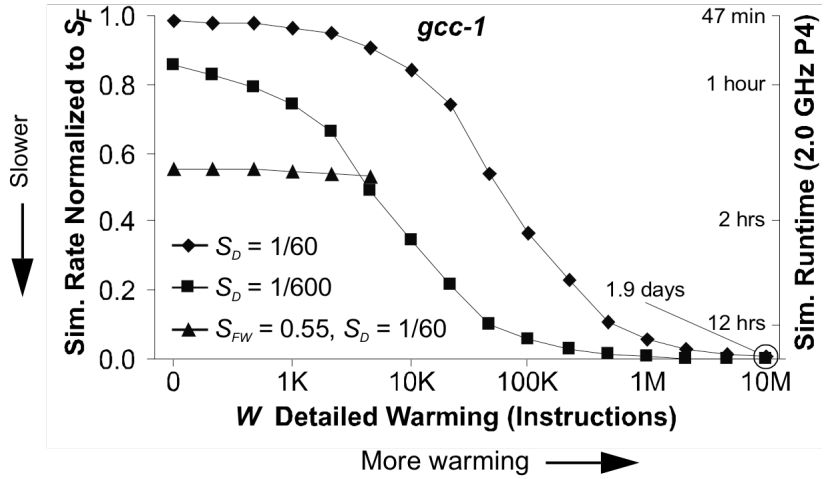


Figure 3.4: Modeled SMARTS simulation rate

The two S_D plots show the simulation rate without functional warming. The S_{FW} plot shows the simulation rate when using functional warming to bound W. The plot shows that when W is greater than approximately 100,000 instructions, functional warming is faster than fast-forwarding because W can be bounded when using functional warming.

The plot shows that SMARTS simulation speed decreases from S_F to S_D as W is increased; furthermore, the anticipated future S_D results in an earlier and sharper decrease. Therefore, unless W can be bounded to a reasonably small value, full benchmark measurement by simulation sampling would remain prohibitively slow.

The simulation rate of SMARTS with functional warming can be derived from the expression for detailed warming by substituting S_{FW} (the functional warming simulation rate) for S_F. Functional warming allows us to bound W to less than a few thousand instructions—sufficiently few such that detailed warming does not affect the simulation rate. This implies that the simulation rate of SMARTS with functional warming stays close to the simulation rate of S_{FW} and is relatively insensitive to the performance of the detailed simulator. In other words, the SMARTS framework enables researchers to apply otherwise prohibitively slow detailed simulators to study complete benchmarks, provided efficient functional warming is possible. In the next section, we will present our implementation of SMARTS where $S_{FW} \approx 0.55$.

3.2 Implementation

To study and demonstrate the effectiveness of the SMARTS framework, we developed SMARTSim, a concrete implementation of a sampling microarchitecture simulator. In this section, we describe the implementation of SMARTSim and revisit the issues of microarchitectural state generation in greater detail. In particular, we explain the effect of detailed warming on the choice of sampling unit size and analyze the effectiveness of detailed warming and functional warming in generating accurate microarchitectural state for sample measurements.

3.2.1 SMARTSim

SMARTSim is built on our enhanced sim-outorder as described in Section 3.1.2. sim-outorder supports a functional simulation mode, similar to the operation of sim-fast in SimpleScalar, that runs approximately 60 times faster than detailed simulation. However, sim-outorder only supports functional simulation prior to starting detailed simulation. SMARTSim allows repeated transitions back-and-forth between functional and detailed simulation modes.

SMARTSim accepts sim-outorder command line arguments and configuration files. In addition, SMARTSim accepts the systematic sampling parameters U, k, W, and j (described in Section 3.1.1). SMARTSim also supports two fast-forwarding options: functional simulation only, and functional simulation with warming (a.k.a. functional warming). For functional warming, SMARTSim performs in-order functional instruction execution and maintains the state of L1/L2 I/D caches, TLBs, and branch predictors in a fashion similar to sim-cache and sim-bpred of SimpleScalar. In SMARTSim, functional warming operations introduce an overhead of approximately 75% over functional simulation alone.

3.2.2 Optimal sampling unit size

SMARTSim allows the user to specify the sampling unit size U. In the analysis in Section 3.1.3, we have shown that smaller unit sizes reduce the number of instructions simulated in detail if the cost of detailed warming is ignored. However, because detailed warming adds an overhead of W instructions of detailed simulation per sampling unit, the optimal value for U increases with increased W to amortize the overhead of detailed warming.

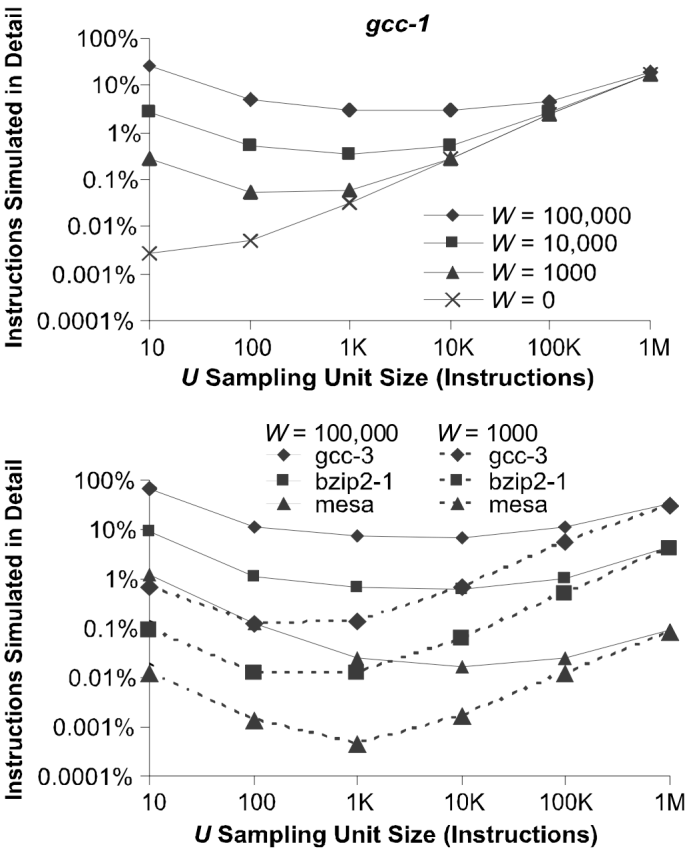


Figure 3.5: Optimal sampling unit size (*U*)

The top chart shows that the optimal *U* increases with detailed warming per measurement (*W*). The bottom chart illustrates that *U* = 1000 instructions is a reasonable choice across benchmarks (extremes and the median are plotted) and *W*.

To illustrate the effect of *W* on the choice of *U*, Figure 3.5 (top) plots the fraction of instructions simulated in detail (i.e., $n(W + U) / N$) for various values of *U* and *W*. The data points are based on SMARTSim execution of gcc-1 on the 8-way configuration, with *n* chosen for 99.7% confidence interval of ±3% in the CPI estimate. In the idealized case where *W* = 0, the minimum *U* leads to the fewest detail-simulated instructions. For non-ideal *W*, however, the optimal value of *U* lies in the range of 100 to 10,000 instructions. Figure 3.5 (bottom) locates the optimal values of *U* for three other benchmarks, gcc-3, bzip2-1, and mesa. Each benchmark is plotted for two values of *W* (1000 and 100,000) that are approximately the magnitudes needed for sampling with and without functional warming, as discussed in the

following two sections. The optimal choice of U is not fixed across benchmarks. However, in all cases, including other SPEC CPU2000 benchmarks not shown, fixing U to 1000 leads to a sufficiently small fraction of detail-simulated instructions such that choosing the optimal U gains at most tens of minutes in SMARTSim run time. Therefore, we suggest using $U = 1000$ in all cases.

3.2.3 Effectiveness of detailed warming

Microarchitectural state can always be warmed to an arbitrary degree of accuracy given sufficient detailed warming. Unfortunately, the required amount of detailed warming to obtain a given degree of accuracy cannot be determined analytically. The required amount is a function of both the benchmark behavior and the microarchitectural mechanisms involved. As a rule of thumb, we expect the amount of detailed warming to scale with the size of the microarchitectural state; however, there are counter-examples.

Table 3.4: Microarchitecture warming requirements without functional warming

The warming requirements of SPEC CPU2000 vary widely and unpredictably. Functional warming removed the need to predict warming requirements for new benchmarks.

W to achieve < 1.5% bias	Benchmarks
$W \leq 50 \times 10^3$	applu, apsi, art-1, art-2, eon-1, eon-2, equake, fma3d, gzip-1, gzip-2, gzip-3, gzip-4, lucas, mesa, sixtrack, twolf
$W \leq 250 \times 10^3$	crafty, eon-3, gap, gcc-1, gcc-3, gcc-4, mcf, swim, vortex-3, vpr
$W \leq 500 \times 10^3$	ammp, bzip2-1, bzip2-2, galgel, gcc-2, gcc-5, gzip-5, vortex-1, vortex-2
$W > 500 \times 10^3$	bzip2-3, facerec, mgrid, parser, perlbmk, wupwise

To better understand the requirements of detailed warming (unaided by functional warming), we experimentally determine the minimum acceptable value of W for the benchmarks with the 8-way configuration such that the bias due to residual microarchitectural state error is just below ±1.5%. (We choose $U = 1000$ and n sufficient for a 99.7%

confidence interval of ±3%.) In systematic sampling, the true bias is the average error over all k possible systematic samples. Exact determination of bias is prohibitively expensive, since k is typically on the order of 10,000 in this study. Therefore, we approximate the procedure by averaging the errors of 5 evenly distributed systematic sampling runs (i.e., $j = \{0, k/5, 2k/5, 3k/5, 4k/5\}$). Table 3.4 categorizes the studied benchmarks according to their required values of W.

Without functional warming, the required W varies widely across benchmarks and inputs. Many benchmarks are insensitive to the accuracy of microarchitectural state, requiring less than 50,000 instructions of detailed warming per measurement period. For some benchmarks, however, even W = 500,000 results in unacceptable bias, as high as 25% for mgrid.

With the exception of the benchmarks requiring more than 500,000 instructions of detailed warming, detailed warming does not significantly impact the simulation rate of SMARTSim. Even 500,000 instructions warmed per sampling unit is a small fraction of the full benchmark. Nevertheless, Table 3.4 does highlight a key shortcoming of the detailed-warming-only approach: the unpredictability of W. Our empirical determination of W is impractical because it requires *a priori* knowledge of the true unbiased CPI derived from prohibitively time-consuming detailed simulation of complete benchmarks.

3.2.4 Bounded detailed warming

Functional warming helps redress the unpredictability of W in detailed warming. Functional warming of problematic microarchitectural state allows us to bound W safely for the remaining state by analyzing the details of the microarchitecture model. For example, to estimate CPI, W needs to be chosen such that an instruction's latency cannot be influenced

by unwarmed microarchitectural state. This requires *W* to exceed the maximum instruction stream distance that latency-influencing state can propagate.

An instruction can only affect the latency of another instruction if there is some history of the former still present at the time the latter is fetched. Outside of long-term architectural (register, memory, etc.) and microarchitectural state (cache, TLB, branch predictor, etc.) maintained by functional warming, the effects of an instruction are bounded by the instruction's lifetime in the microprocessor. With the exception of store instructions, when an instruction commits, its associated short-term state is freed. A committed store instruction that misses in the cache might stall a later store instruction by causing the store buffer to overflow. Hence, a worst-case bound on *W* is the product of store-buffer depth, memory latency in cycles, and the maximum IPC. For our 8-way configuration, this upper bound is 12,800 ($16 \times 100 \times 8$) instructions. In practice, this worst-case behavior does not occur; all the 8-way results presented in this study were achieved with only 2000 instructions of detailed warming, and 16-way results with 4000.

3.2.5 Effectiveness of functional warming

Even with both functional and detailed warming, some inaccuracies in microarchitectural state remain and contribute to errors in the estimates as bias. Table 3.5 reports the residual bias in the CPI estimated by SMARTSim when functional warming is employed in conjunction with detailed warming of the aforementioned values of *W*. Benchmarks are presented in sorted-order by the worst bias. All benchmarks have bias under ±2.0% and only 6 benchmarks in each configuration exceed ±1.0%. The bias is predominantly due to wrong-path and out-of-order effects in caches and the branch predictor. This set of results corroborates our conclusion that functional-warming with bounded *W* is effective in reducing microarchitectural state warming bias.

Table 3.5: CPI bias with functional warming and minimal detailed warming

Detailed warming of a few thousand instructions is sufficient to reduce bias to acceptable levels for all SPEC CPU2000 benchmarks.

8-way $W = 2k$	CPI Bias	EPI Bias	8-way	CPI Bias	EPI Bias	16-way $W = 4k$	CPI Bias	16-way	CPI Bias
vpr	-1.56%	0.52%	vortex-1	-0.29%	0.80%	mcf	1.88%	gzip-1	-0.25%
galgel	1.37%	0.04%	gcc-4	-0.29%	-0.09%	gcc-2	-1.60%	galgel	-0.25%
gcc-2	-1.07%	0.63%	mgrid	0.28%	0.09%	vortex-3	1.18%	gcc-4	0.24%
bzip2-2	-1.04%	0.94%	bzip2-1	-0.25%	0.55%	eon-2	-1.11%	bzip2-2	-0.19%
parser	1.01%	0.56%	gzip-3	-0.25%	0.05%	gcc-5	-1.10%	mgrid	-0.17%
gzip-5	0.94%	2.31%	ammp	0.18%	-2.42%	sixtrack	-0.93%	art-1	-0.15%
facerec	0.86%	0.96%	sixtrack	0.17%	-0.27%	wupwise	0.85%	gzip-2	0.14%
gcc-5	-0.81%	0.04%	wupwise	-0.17%	-0.05%	bzip2-1	0.78%	gzip-5	-0.12%
vortex-3	-0.55%	0.63%	equake	0.13%	1.46%	applu	0.65%	vortex-2	-0.12%
gcc-1	-0.53%	-1.03%	applu	-0.12%	-0.04%	mesa	-0.58%	lucas	0.09%
bzip2-3	-0.51%	0.36%	gzip-4	-0.11%	0.09%	eon-1	-0.56%	art-2	0.07%
perlbmk	-0.40%	-0.09%	eon-2	-0.10%	-0.10%	vortex-1	-0.54%	apsi	-0.07%
swim	0.38%	0.19%	twolf	0.09%	0.00%	ammp	-0.53%	parser	0.06%
gzip-1	0.38%	1.43%	gzip-2	-0.09%	0.23%	swim	0.44%	gzip-3	-0.05%
mcf	0.36%	1.14%	mesa	-0.07%	0.11%	vpr	0.38%	twolf	0.04%
eon-1	-0.36%	-0.14%	gap	0.07%	2.49%	gcc-3	-0.36%	bzip2-3	-0.04%
fma3d	-0.35%	-0.22%	gcc-3	-0.05%	0.00%	crafty	0.32%	eon-3	0.04%
crafty	-0.35%	-0.14%	eon-3	-0.04%	-0.15%	perlbmk	-0.30%	facerec	0.03%
art-2	0.31%	0.13%	lucas	0.03%	0.13%	fma3d	0.28%	equake	0.02%
art-1	-0.30%	-0.40%	vortex-2	0.02%	1.09%	gap	0.28%	gzip-4	0.00%
apsi	0.29%	-0.42%				gcc-1	0.25%		

3.3 Results

This section outlines an exact procedure for estimating a target metric using statistical simulation sampling. We evaluate the effectiveness of this procedure by estimating the CPI and energy per instruction (EPI) of SPEC CPU2000 using SMARTSim.

3.3.1 SMARTS procedure

One iteration of a SMARTS measurement run requires the user to supply three sampling simulation parameters: *W*, *U*, and *n*. First, *W* is selected to exceed the bounded history of the microarchitectural state as described in Section 3.2.4. We recommend utilizing functional warming (see Section 3.2.5) whenever possible, as it greatly simplifies the determination of

W. Our 8-way results were achieved with $W = 2000$ instructions, and 16-way results with $W = 4000$. Second, we suggest setting $U = 1000$. We have shown in Section 3.2.2 that $U = 1000$ is appropriate for all SPEC CPU2000 benchmarks. Lastly, we elaborate on how to determine *n*, and correspondingly *k*, to meet a desired confidence in the following paragraphs.

In general, the correct value for *n* must be determined in a two-step process. First, a sampling measurement is made using a generic initial value n_{init} that is a compromise between simulation rate and the likelihood of meeting the confidence requirement on the first try. If the choice of n_{init} is shown to be insufficient after one sampling simulation, a second step is required where n_{tuned} for a second sample is calculated from the $\hat{V}_x$ of the initial run.

A priori, the minimum value of *n* to achieve a given confidence is unknown for an arbitrary benchmark and simulated microarchitecture. Given a fixed confidence target, *n* must be adjusted according to the coefficient of variation V_{CPI} of the population. Based on our analysis of V_{CPI} of SPEC CPU2000 benchmarks (in Section 3.1.3), we conjecture that the values of V_{CPI} tend to cluster around 1.0 for most benchmarks and simulated microarchitectures when $U = 1000$. Hence, from $n_{init} = (z/\varepsilon)^2$, we infer that $n_{init} = 10{,}000$ is likely to yield 99.7% confidence interval of ±3%. Given $N = 9{,}420{,}910$ for the smallest of our SPEC CPU2000 benchmarks, $n_{init} = 10{,}000$ still represents a very small fraction of detail-simulated instructions and hence has minimal impact on simulation turnaround time.

One run of SMARTS measurement with $k = N/n_{init}$ produces an initial estimate of average CPI and $\hat{V}_{CPI}$ of the sample. Because the confidence of an estimate is jointly quantified by the two interdependent terms confidence level $(1 - \alpha)$ and confidence interval $\pm\varepsilon \cdot \overline{X}$, one can

either set a desired confidence level and calculate the obtained confidence interval for a given sample, or vice versa. For a set confidence level $(1 - \alpha)$, the confidence interval is $\pm(z \cdot V_x \cdot \bar{x})/\sqrt{n}$ where z is the $100[1 - (\alpha / 2)]$ percentile of the standard normal distribution. Commonly used confidence levels are 95% and 99.7% (a.k.a. 3σ or virtually-certain). Corresponding values of z are 1.97 and 3, respectively. If the confidence level and interval yielded by the initial sample are unacceptable, the n_{tuned} to achieve a desired confidence on the next sample is $((z \cdot \hat{V}_x)/\varepsilon)^2$. If the initial confidence is overly below target, we suggest slightly overestimating n_{tuned} for the subsequent run. In any case, the actual confidence achieved by the subsequent sample must be checked using the subsequent sample's new $\hat{V}_{CPI}$.

The above treatment of confidence considers only the error introduced by statistical sampling. In practice, the true error margin in an estimate must also account for any bias in the measurements. Recall from Chapter 2 that if the bias is known, it can be accounted for by subtracting it from the estimate, without affecting confidence. If the bias can only be bounded, then it introduces a proportional amount of uncertainty in the estimate beyond the confidence interval.

3.3.2 Performance and accuracy

We applied the procedure outlined above to SPEC CPU2000 benchmarks using SMARTSim. Figure 3.6 reports results of CPI estimated using SMARTSim in one run with n_{init} = 10,000. Benchmarks are shown in sorted order by worse confidence intervals. For each benchmark, we show the actual achieved error and the predicted confidence interval calculated from $\hat{V}_{CPI}$ for 99.7% confidence. The confidence interval accounts for random error in the estimated CPI that is introduced by systematic sampling. Notice that actual error resulting

from 10,000 sampling units is generally much less than the predicted confidence interval. A large part of this error can be attributed to the residual bias of imperfect microarchitectural state warming (functional warming with fixed *W*), with only a very small component caused by statistical sampling.

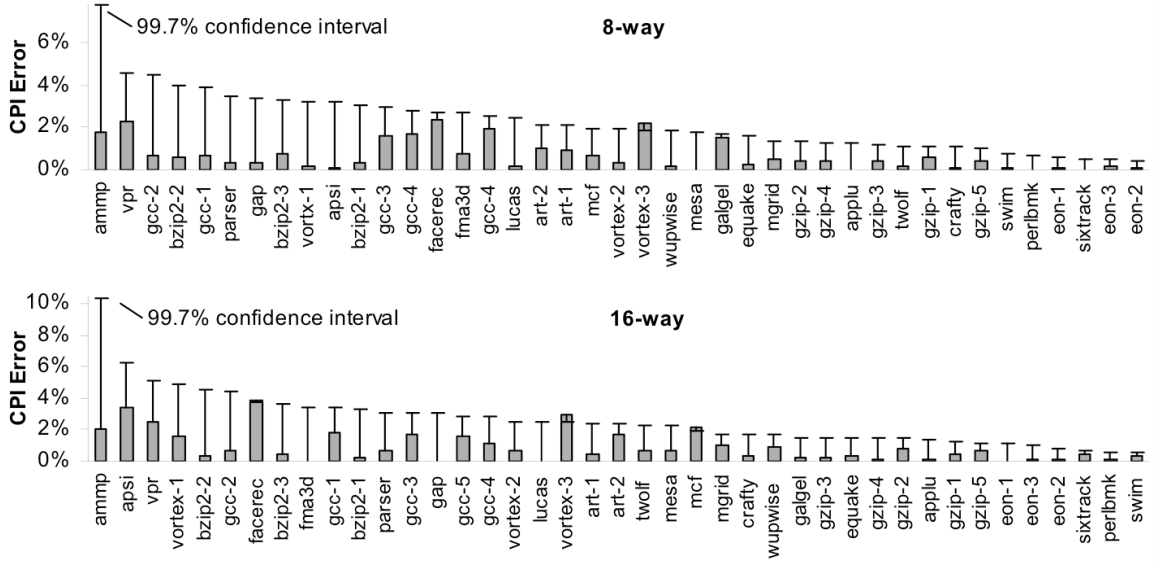


Figure 3.6: SMARTS CPI results with *n* = 10,000 instructions

Unacceptably large confidence intervals (e.g., 8-way ammp, vpr, and gcc-2) can be improved by simulating with n_{tuned}.

For most of the benchmarks, n_{init} achieves a confidence interval within ±3%. For benchmarks with confidence intervals greater than ±3%, simulation sampling needs to be repeated using n_{tuned}—calculated from the $\hat{V}_{CPI}$ of the initial sample. For example, rerunning simulations for the 8-way configuration with n_{tuned} of 66,531 (ammp), 23,321 (vpr), and 21,789 (gcc-2) achieve actual errors of 1.1%, 0.1%, and –0.9% with confidence intervals of 3.0%, 2.9%, and 2.6%. To this confidence interval, we add an uncertainty due to microarchitectural state warming bias, which we empirically bound to below 2%.

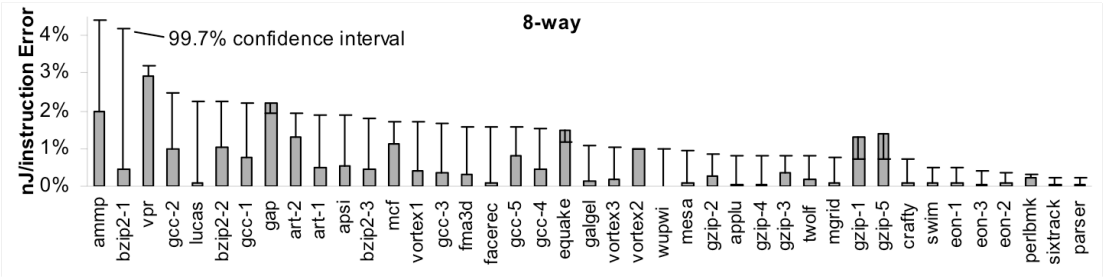


Figure 3.7: SMARTS EPI results with *n* = 10,000 instructions

V_{EPI} tended to be lower than V_{CPI}, leading to tighter confidence intervals when sampling EPI.

Figure 3.7 presents the results of applying SMARTS to estimating energy per instruction (EPI). As in CPI estimations, we find in most cases initial sampling simulations using n_{init} = 10,000 achieves confidence intervals tighter than ±3%. Confidence intervals for EPI estimation tend to be tighter than CPI confidence intervals because of less variability in EPI. Unfortunately, the smaller predicted confidence intervals are overshadowed by the microarchitectural state warming bias. With the exception of gap, equake, and gzip, the actual errors are within the confidence interval. For these exceptions, we have determined experimentally that the error is almost entirely due to bias as shown in Table 3.5.

Table 3.6 compares simulation runtimes for functional (i.e., sim-fast), detailed (i.e., sim-outorder with detailed memory models), and SMARTSim simulation on a 2 GHz Pentium 4. SPEC CPU2000 benchmarks on the 8-way configuration are shown in sorted by length in instructions. As shown in Table 3.6, detailed simulation takes on average 7.2 days and can take as long as 23 days. In contrast, SMARTSim takes on average 5.0 hours and in the worst-case slightly less than 16 hours. SMARTSim simulation speed is around 50% of functional-only simulation for most microarchitecture configurations.

Table 3.6: SMARTS runtimes compared to detailed and functional simulation

Runtime (hrs.)	Detailed	Functional	SMARTS	Runtime (hrs.)	Detailed	Functional	SMARTS
parser	541	9.2	15.8	bzip2-3	123	2.1	3.6
sixtrack	466	7.9	13.6	vortex-1	118	2.0	3.5
mgrid	414	7.0	12.1	bzip2-1	108	1.8	3.2
galgel	405	6.9	11.8	gcc-2	107	1.8	3.2
wupwise	346	5.9	10.1	gzip-3	103	1.7	3.0
apsi	344	5.8	10.1	eon-1	100	1.7	2.9
twolf	343	5.8	10.0	gzip-1	83	1.4	2.4
ammp	323	5.5	9.6	vpr	83	1.4	2.5
mesa	278	4.7	8.1	gzip-4	81	1.4	2.4
gap	266	4.5	7.8	eon-2	80	1.4	2.3
fma3d	265	4.5	7.8	gcc-5	61	1.0	1.8
swim	223	3.8	6.5	mcf	61	1.0	1.8
applu	221	3.8	6.5	eon-3	57	1.0	1.7
facerec	209	3.5	6.1	gcc-1	46	0.8	1.4
crafty	190	3.2	5.5	art-2	45	0.8	1.3
gzip-5	167	2.8	4.9	art-1	41	0.7	1.2
bzip2-2	142	2.4	4.2	gzip-2	39	0.7	1.2
lucas	141	2.4	4.1	gcc-4	13	0.2	0.4
vortex-2	137	2.3	4.0	gcc-3	12	0.2	0.4
vortex-3	132	2.2	3.9	perlbmk	2	0.1	0.1
equake	130	2.2	3.8	mean	171.8	2.9	5.0

3.3.3 Comparison to SimPoint

SimPoint [Hamerly et al. 2005], also enables reduced simulation turnaround time. SimPoint selects representative subsets of benchmark traces via offline analysis of basic blocks. Using clustering algorithms, SimPoint selects and weights several large sampling units such that the frequency of each static basic block across the weighted units matches that block's frequency in the full dynamic stream. A fundamental assumption of SimPoint is that all dynamic instances of basic block sequences with similar profiles have the same behavior, therefore a particular sequence can be measured once and weighted appropriately to represent all remaining instances [Lau et al. 2005].

SimPoint has two key advantages: (1) due to large sampling units, SimPoint obviates the need for functional warming and can be more quickly integrated into a simulation infrastructure, and (2) SimPoint allows early termination of simulation after all selected sections have been visited.

We implemented SimPoint with our SimpleScalar toolset and verified our implementation against the published configuration and results in the first SimPoint work [Sherwood et al. 2002]. This work recommended up to ten 100M-instruction sampling units. SimPoint resulted in an average improvement of 1.8 times in simulation speed over SMARTS for our 8-way configuration. An updated procedure and software release for SimPoint, version 3.0, now recommend approximately thirty 10M-instruction sampling units and use an improved clustering heuristic. The 10M-instruction sampling units and improved clustering results in half the error on average, and reduce the amount of detailed simulation by roughly three times [Hamerly et al. 2005].

However, SimPoint has several shortcomings: (1) it may result in arbitrarily high CPI error, (2) it does not offer quantifiable confidence in estimates, and (3) it does not allow trading off confidence in results for speed which becomes a limitation when using checkpoints instead of fast-forwarding (see Section 4.4).

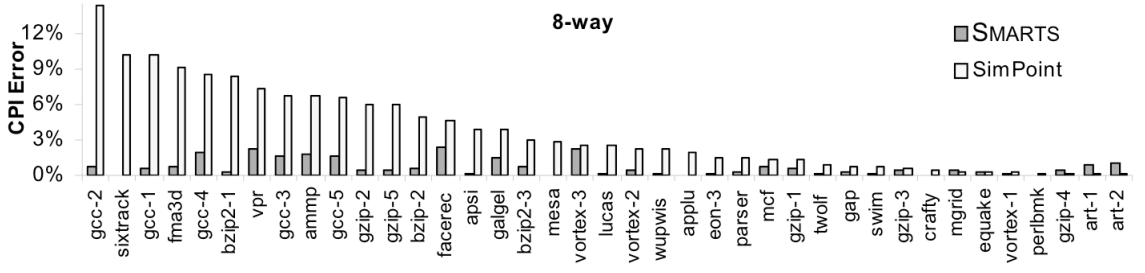


Figure 3.8: Comparison of SMARTS with SimPoint

SimPoint's mean runtime per benchmark is 2.8 hours compared to 5.0 hours for SMARTS.

Comparison of SMARTS with SimPoint presents a comparison of CPI error between SimPoint and SMARTS for the benchmarks presented in [Sherwood et al. 2002] running on our 8-way configuration. The comparison shows that SimPoint has a higher average error (3.7% vs. our 0.6%) and considerably higher worst-case error (-14.3% for gcc-2).

Gcc-2 is an example where SimPoint produces an unacceptably high CPI error when running on our 8-way configuration. However, simulation using the published microarchitecture configuration in [Sherwood et al. 2002] only results in a 1.6% error. In gcc-2, we observed that the program phases chosen by SimPoint to be measured with a single sampling unit exhibit large variations in their L2 miss rate. The large variations within each phase results from too few sampling units being selected by SimPoint to measure all the different behaviors of gcc-2. Different behaviors were clustered together, and only one measurement was taken per phase to represent these diverse behaviors. Therefore, in this case, the SimPoint estimate based on few large sampling units yields a large error. In contrast, independent of benchmark and microarchitecture configuration, SMARTS uses the measured coefficient of variation to help gauge both the required sample size and the confidence in the estimates.

3.3.4 Beyond SPEC CPU2000

While SPEC CPU2000 is still the most widely used suite of general purpose CPU benchmarks, there are many other benchmarks used in the computer architecture community. In addition, SPEC CPU2006 contains applications with longer runtimes and larger memory footprints. We expect these benchmarks to require minimal, if any, changes to the SMARTS framework.

SMARTS has three parameters that should be revisited when measuring a new benchmark, *W*, *U*, and *n*. The procedure for selecting these parameters is presented in Section 3.3.1. The detailed warming interval length, *W*, is a function of the microarchitecture under study, and may increase with more complex architectures but will not vary with new benchmarks. Benchmarks longer than SPEC CPU2000 do not cause an increase in sample size, *n*; only an increase in performance variability (e.g., V_{CPI} and V_{EPI}) will require larger sample sizes. We do not expect performance variability to increase markedly with new benchmarks. Finally, the optimal value for the sampling unit size, *U*, is governed mostly by the magnitude of *W*, and the rate of change in performance variability across potential sampling unit sizes (e.g., Figure 3.2). We expect that *U* = 1000 or *U* = 10,000 instructions will continue to be optimal or near optimal for most other benchmarks.

3.4 Related work

This study investigated the optimal sampling parameters for microarchitecture simulation. The SMARTS framework also prescribes an effective warming strategy that supports many small measurements. The framework's combination of sample design and practical implementation produce highly accurate and reliable results for timing-accurate microarchitecture simulation. There is a large volume of previous work on performance simulation sampling that we extend with our tuning of sampling parameters and our design of an appropriate warming technique for a contemporary simulator and benchmark suite. Note that simulation sampling is distinct from analytic modeling approaches often referred to as statistical simulation [Eeckhout et al. 2003].

Much of the early work in simulation sampling was performed in the context of trace-based simulators [Smith 1982]. The inputs used by these simulators were traces like

memory access or branch direction data captured from real machines or functional emulators. Only portions of a microarchitecture can be simulated from such traces, and thus trace-based simulators were used in studies of stand-alone components such as caches and branch predictors. Trace-based simulators generally cannot be used to estimate runtime on modern CPUs. However, these simulators can easily sample a recorded trace without the overhead of fast-forwarding between measurements. The ability to quickly seek to any part of a trace leaves only a sample design and a warming strategy that eliminates cold-start bias [Easton and Fagin 1978] to be devised.

Kessler, Hill, and Wood [1994] performed a comprehensive survey and comparison of memory access trace-sampling techniques. One of the two evaluated sample designs was set sampling, an approach specific to cache simulation. The second sample design, time sampling, systematically sampled contiguous groups (a sampling unit) of cache accesses [Laha, Patel, and Iyer 1988]. Five warming techniques were compared across several sampling unit sizes (sample size was fixed to 30 measurements): *cold* assumed an empty initial cache; *half* warmed an empty cache for the first half of a sampling unit, and measures performance for the second half; *prime* measured only fully-warmed cache sets [Laha, Patel, and Iyer 1988]; *stitch* preserved the cache state between sampling units [Agarwal, Hennessy, and Horowitz 1988]; and *initmr* used Wood et al.'s [1991] analytic model to estimate cold-start miss rates. The resulting bias of these warming approaches was compared for 1, 4, and 16 MiB caches with sampling unit sizes of 0.1, 1, 10, and 100 million instructions. The *initmr* warming approach was found to be the least biased on average; it produced less than 10% bias in miss rate for two-thirds of the tested cases.

Completing the warming phase before the start of measurement (like the *half* technique) can decouple the amount of warming from the sample unit size. Haskins and Skadron

[2003] propose an analysis to probabilistically determine the amount of warming required before a measurement to ensure a warmed cache. An amount of warming is determined for each sampling unit based on the distribution of memory reference reuse latencies (MRRL) of the instruction stream prior to the sampling unit. Haskins and Skadron recommend the 99.9^{th} percentile of reuse latencies, in terms of instruction count, for warming. Experimental results show very low bias in estimated IPC, however the analysis of MRRL was performed with large sampling units (1 million instructions) that can amortize bias, and no direct comparison to *initmr* was done. MRRL requires a functional simulation of the entire benchmark before producing warming requirements output.

We investigated applying MRRL to the SMARTS framework and found double the bias as functional warming, 1.1% on average as compared to 0.6% (Section 4.1.4). In addition, MRRL required tens of millions of instructions of warming for each 1000 instruction sampling unit on average.

A more recent work in the same vein as MRRL is boundary line reuse latency (BLRL) by Eeckhout et al. [2005] which considers only the reuse latencies that cross the boundary line between the region before a sampling unit and the sampling unit to compute warming requirements. BLRL achieves approximately the same bias of MRRL with half the warming requirements.

4 Checkpoint sampling

Functional warming is the main performance bottleneck of simulation sampling and requires hours of runtime while the detailed simulation of the sample requires only minutes. Existing simulators can avoid functional simulation by jumping directly to particular instruction stream locations with architectural state checkpoints. To replace functional warming, these checkpoints must additionally provide microarchitectural model state that is accurate and reusable across experiments while meeting tight storage constraints.

This chapter describes our simulation sampling framework that replaces functional warming with live points without sacrificing accuracy. A live point stores the bare minimum of functionally-warmed state for accurate simulation of a limited execution window while placing minimal restrictions on microarchitectural configuration. Live points can be processed in random rather than program order, allowing simulation results and their statistical confidence to be reported while simulations are in progress.

4.1 Implementation

In this section we present our approach for creating small, fast-loading, independent, reusable, and accurate checkpoints to enable effective sampling.

4.1.1 Methodology

We evaluate live points in a sampling simulator based on the SimpleScalar 3.0 sim-outorder simulator [Burger and Austin 1997] for the Alpha ISA. We modify sim-outorder's memory subsystem to include a store buffer and miss status holding registers (MSHRs), and model

interconnect bottlenecks in the memory hierarchy. We encode live points using ASN.1 DER format [International Organization for Standardization 2002] and gzip compression, which incur minimal storage and processing time overhead. We use all 26 SPEC CPU2000 benchmarks [Henning 2000] and evaluate all reference inputs except vpr-place and three perlbmk inputs, as these inputs fail to simulate correctly in sim-outorder. Overall, we include 41 benchmark/input set combinations in this study.

Without loss of generality, we use CPI (cycles-per-instruction) as our target metric for estimation. We measure CPI bias by averaging actual error (relative to full sim-outorder simulations) over five different samples, according to the methodology described in Section 3.3.1.

We evaluate live points with two microarchitectural configurations. Our baseline 8-way out-of-order superscalar model represents a processor in the current technology generation. The 16-way out-of-order superscalar configuration is included to reflect an aggressive future design point. This configuration has a wider data path, larger out-of-order window, and larger caches, to exercise the effects of enlarged microarchitectural state. The details of the 8-way and 16-way configurations are summarized in Table 3.3.

We use the sampling approach from Section 3.3.1., periodic 1000 instruction measurement intervals, to identify measurement locations for all experiments. This sample design has been demonstrated to minimize the total number of instructions in detailed windows, and thus, detailed simulation time. However, live points can also be applied to other sample designs (e.g., random sampling). We choose sample size to achieve precisely 99.7% confidence of ±3% error for each result. We report simulation runtimes for systems with 2.80 GHz Intel Xeon (512 KiB L2) processors.

4.1.2 Why checkpointed warming

Functional warming repeats architectural state updates across different simulations of the same benchmark. (Simulating workloads for which architectural state varies across repeated runs—i.e., because of interrupt timing or different interleaving of multiprocessor instruction streams—is beyond the scope of this work.) Frequently, microarchitectural state updates are also identical across runs. Checkpoints can memoize the redundant calculation across runs, amortizing the one-time cost of computing warmed state. We are interested in finding the best way to take advantage of checkpoints to accelerate warming.

Although some microarchitecture studies have suggested or used checkpoints to accelerate simulation [Barr et al. 2005; Ekman and Stenström 2005; Girbal et al. 2003; Perelman, Hamerly, and Calder 2003], none have explored the space of microarchitecture warming solutions in the context of checkpointing. For each portion of model state generated by functional warming, we may choose either to construct the state dynamically, or store it in checkpoints. This choice impacts simulation sampling along three dimensions: the accuracy of the warmed state, the reusability of checkpoints across microarchitectural configurations, and the speed of simulation. In this section, we explore the warming method design space with respect to these three dimensions and justify our choice of checkpointed warming to implement live points.

4.1.3 Simulation sampling warming methods

There is a rich design space of possible warming strategies that combine checkpoints and dynamic warming for various portions of architectural and microarchitectural model state. We restrict our exploration to strategies that use detailed warming to initialize queue and pipeline state. Detailed warming can reconstruct state for the vast majority of microarchi-

tectural structures rapidly, and the amount of required warming can be determined via worst-case analysis. By warming most structures dynamically, we avoid storing any state for these structures, and do not constrain model parameters that affect this state.

Evaluation criteria. We focus our design exploration on warming alternatives for long-history structures, such as caches and branch predictors, for which detailed warming is prohibitively slow. We evaluate alternatives based on their accuracy, checkpoint reusability, and speed.

With respect to accuracy, we consider only the bias introduced by the warming strategy. SMARTS demonstrated low bias—0.6% on average, 1.6% worst case—using functional warming. It is essential to maintain this high accuracy when accelerating warming because we cannot detect bias through statistical confidence calculations.

We evaluate the reusability of a warming methodology in terms of the restrictions it places on simulator configuration. When we store the warmed state of microarchitectural structures in a checkpoint, we may be forced to limit some of the configuration parameters for that structure.

Finally, we evaluate the speed of warming alternatives in two ways. First, we consider how fast measurements can be processed. For all alternatives, time to simulate the detailed window is the same, while functional warming and checkpoint decompression/loading time varies. Second, we consider whether detailed windows are independent, or must be simulated in program order. Independent windows can be simulated in parallel, and enable online reporting of measurement results.

Warming methods. Figure 4.1 depicts alternatives in the warming strategy design space. At one extreme, functional warming is used for the entire duration between measurements,

without checkpoints (as in SMARTS). We refer to this method as full warming. The opposite extreme, checkpointed warming, eliminates all functional warming and stores long-history state in checkpoints. This approach requires limiting some design parameters of the checkpointed structures.

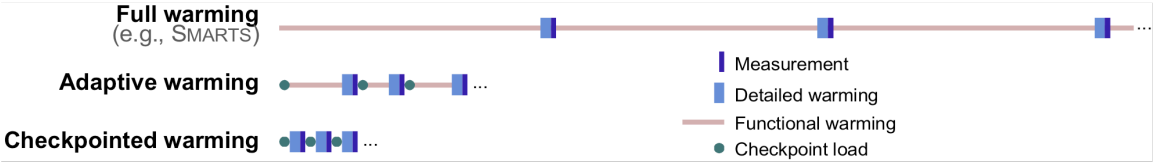


Figure 4.1: Simulation sampling warming methods

All methods use the same sample design and confidence intervals, only bias differs.

Functionally-warming microarchitectural state for the entire duration between measurements is usually not necessary. In adaptive warming, we store architectural state in checkpoints, and reconstruct long-history state with a reduced functional warming period. Adaptive warming requires a mechanism to determine precisely how little functional warming each detailed window requires.

Trade-offs. Figure 4.2 illustrates the relationship between each warming alternative and our three evaluation criteria. Each alternative optimizes for two of the design criteria (the two depicted nearest it), at the expense of the third.

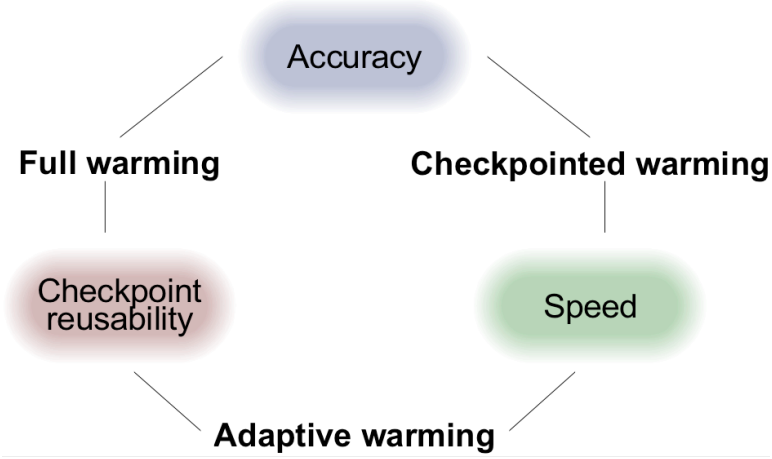


Figure 4.2: Relative merits of warming methods

Full warming maximizes accuracy and flexibility, but its need for long periods of functional warming makes it slow, and its turnaround time scales with benchmark length. As full warming requires no checkpoints, no configuration parameters are fixed.

Adaptive warming maintains the reusability of full warming and improves speed, but we show that it sacrifices accuracy. The accuracy and speed of adaptive warming depend on a rigorous determination of the minimal functional warming period for each detailed window. Unfortunately, determining the correct amount of warming remains a difficult and unsolved problem [Kessler, Hill, and Wood 1994].

Checkpointed warming matches the accuracy of full warming and maximizes speed, at the expense of checkpoint reusability. Checkpointed warming achieves this accuracy because it uses full warming simulation to generate the checkpointed state.

Because checkpointed warming spends no time performing functional warming, it is the fastest alternative. The drawback of checkpointed warming is that it imposes limits on some aspects of the simulated microarchitectural parameters (e.g., the maximum size or associativity of a cache), which constrains checkpoint reusability. Reusability is important because

we must amortize the one-time cost of checkpoint creation (roughly the cost of a full-warming simulation) over a series of experiments.

Each of the three warming approaches suffers from a different key weakness. The speed of full warming has been quantified in Section 3.1.4. We evaluate the accuracy of adaptive warming in Section 4.1.4. We then explore the reusability of checkpointed warming in Section 4.1.5.

4.1.4 Adaptive warming

The key challenge of achieving accuracy with adaptive warming lies in determining the functional warming period length. If the warming period is underestimated, simulation results will be biased. If the warming period is overestimated, we sacrifice simulation speed.

A recently-proposed technique for determining cache warming requirements is Memory Reference Reuse Latency (MRRL) [Haskins and Skadron 2003]. MRRL collects a histogram of memory access reuse distances between each pair of detailed windows during a functional simulation of a benchmark. The warming length reported by MRRL is the length sufficient to cover 99.9% of the observed reuse distances. This probabilistic bound on cache warming requirements is configuration independent, because reuse latency is measured by instruction count in a functional simulator. The MRRL analysis outputs specific warming lengths (in instructions) for each detailed window, and must be run once per benchmark and sample design. The offline analysis pass takes roughly the same time as a full-warming simulation.

MRRL has demonstrated low bias on large detailed windows (worst-case error of 2% for 50 million instruction windows). This paper evaluates MRRL on the small detailed

windows required by the optimal sample design. Small windows are more susceptible to bias because warming errors are not amortized over a large measurement interval.

We evaluate MRRL with a reuse probability of 99.9% as recommended in [Haskins and Skadron 2003]. This reuse probability results in an average of 4.1 million instructions of warming prior to each detailed window, which is 20% of the average full warming interval (20.5 million instructions). Thus, an approximation for the runtime of the adaptive warming strategy is 20% of the functional warming time of SMARTS, plus detailed simulation time, or about 1.5 hours on average per benchmark (8-way).

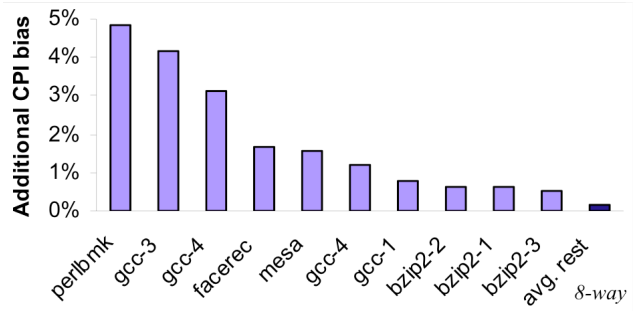


Figure 4.3: Adaptive warming bias

Additional error is introduced by adaptive warming using MRRL vs. full warming.

We present the results of our accuracy evaluation of adaptive warming with MRRL for small windows in Figure 4.3. Both average (1.1%) and worst-case error (5.4%) are considerably worse than full warming (0.6% on average; 1.6% worst-case). Error is high because short detailed windows are sensitive to accurate cache state.

MRRL does not allow detailed windows to be simulated independently because cache state must be stitched [Kessler, Hill, and Wood 1994] between consecutive windows. To obtain low bias, detailed windows must be simulated in program order, precluding parallelization and online result reporting (see Section 4.2). If MRRL is used without stitched

state (thereby assuming an empty cache at the start of each functional warming period) we observe a considerably higher CPI bias of 1.9% on average, with a worst case of 11%.

Because of the high worst-case error and relatively modest speedup of adaptive warming, we do not choose adaptive warming to implement live points. Increasing warming over MRRL (or increasing the MRRL reuse probability threshold) will improve accuracy, but further reduces the speed of adaptive warming.

4.1.5 Checkpointed warming

The key concern in evaluating checkpointed warming is the reusability of a set of checkpoints across a series of experiments. Because checkpointed warming uses a full-warming simulation to generate microarchitectural state for large structures, its accuracy is identical to full warming. When the generated live points can be used for at least two experiments, checkpointed warming provides a net speed gain over full warming.

To maximize the reusability of live points, we wish to place as few constraints as possible on microarchitectural configuration. Checkpointed warming dynamically reconstructs the vast majority of microarchitectural structures (e.g., queues, ROB, etc.) through detailed warming. As such, the configurations of these dynamically-warmed structures are not constrained. For the remaining few structures, for which detailed warming requirements are large or cannot be determined (e.g., caches and branch predictors), we store a representation of the structure in each live point. The reusability of a live point library is limited by the flexibility of these representations.

There are two basic approaches to increasing live point reusability. First, we can collect state snapshots for multiple component configurations in a single creation pass. The second, preferable approach is to modify the saved representation such that a range of organiza-

tions can be reconstructed when a live point is loaded. However, we cannot easily apply this adaptable approach to some structures, such as modern branch predictors, and so we must store multiple warmed configurations. Cache-like structures, including the TLB, can typically be stored using adaptable data structures.

Storing multiple configurations. The first approach is straight-forward and effective if the number of configurations of interest is relatively small. The major cost of live point creation is the traversal of the entire benchmark instruction stream. Warming additional copies of a microarchitectural structure incurs a relatively small overhead. If the slowdown is less than a factor of two, it is a net win to collect state for both configurations in a single pass. We recommend this approach for storing branch predictor state.

Storing adaptable warmed state. With cache-like structures, it is possible to exploit the properties of cache replacement algorithms to create a representation of cache state from which one can accurately reconstruct a range of configurations [Hill and Smith 1989]. Barr et al. propose a data structure, called the Memory Timestamp Record (MTR), that records the timestamp of the last access to each cache block during functional warming [Barr et al. 2005]. The MTR allows a simulator to reconstruct a cache hierarchy of arbitrary sizes and associativities assuming least-recently-used replacement and a lower bound on cache block size.

Storing an MTR in each live point enables reusability across nearly arbitrary cache hierarchy organizations, but incurs a storage cost proportional to the application's memory footprint. However, researchers can often place an upper bound on the maximum cache size of interest. For a given maximum size and associativity, we can instead store a timestamp-sorted list of the most recent accesses mapping to each set, referred to as a Cache Set Record (CSR) by Barr et al. [2005]. A CSR requires the same storage as the tag array for the

selected maximum cache size, and allows reconstruction of all smaller and/or less associative caches.

Our analysis of simulation sampling warming methods demonstrates that checkpointed warming is both fast and accurate. The reusability weakness of checkpointed warming can be mitigated through careful planning of microarchitectural state representation. Thus, we choose to use checkpointed warming to implement live points.

4.1.6 Live points with live state

Current publicly-available computer architecture simulators already provide a checkpoint creation and loading capability that allows the simulator to move to a particular program trace location in constant time [Burger and Austin 1997; Magnusson et al. 2002]. These checkpoint implementations store only architecturally-visible system state (i.e., memory, architectural register and peripheral device state). A straightforward approach to implement checkpointed warming is to extend these existing checkpoints with functionally-warmed microarchitectural state as described in Section 4.1.5.

Unfortunately, this straightforward approach is not practical because conventional checkpoints require prohibitive storage, proportional to the total memory footprint of an application (up to 200 MiB for SPEC CPU2000 [Henning 2000]). We measured an average SPEC CPU2000 memory footprint of 105 MiB. Thus, for SMARTS-like samples (~10,000 measurements), conventional checkpoints for all of SPEC CPU2000 require 33 TB of storage (7.2 TB with gzip compression). Sampling optimizations [Ekman and Stenström 2005; Perelman, Hamerly, and Calder 2003] (Chapter 5) reduce this cost by an order of magnitude at best. With these checkpoint sizes, simulations are I/O bound, and checkpointed warming can provide little, if any, speedup over functional warming. It may be possible to save space

by storing only changes to memory between checkpoints, but this approach introduces dependence among checkpoints, precluding parallel simulation and other sampling optimizations (see Section 4.2).

Reducing storage with live state. We can drastically reduce checkpoint storage cost for live points by storing only the state that will be accessed during the brief simulation window, an approach we call live state. Because the detailed windows are just a few thousand instructions, only a tiny subset of state is accessed. Simulation state that is never referenced during measurement or detailed warming can be omitted from the checkpoint without affecting the simulation.

The live state approach stores the minimal set of accessed state for each live point's specified simulation window. Live points can accurately simulate only the instructions within this pre-selected window. The restriction to a pre-selected window does not impact simulation sampling because the window locations and measurement/detailed warming periods are specified in advance by the sample design.

We can identify precisely which instructions will commit during the selected window when we construct a live point. Thus, it is straightforward to identify all the memory and microarchitectural state these instructions will access—generally less than 32 KB per live point (uncompressed, including ASN.1 encoding overhead).

However, we cannot identify the state that is accessed on non-committed speculative paths (wrong-path instructions). It is not possible to identify a priori the set of wrong-path instructions that will execute in all future simulations at live point creation time. To do so requires either fixing all simulation parameters (queue sizes and latencies), or exploring all possible speculative paths to the depth they might be followed (as bounded by, for example,

ROB size). The former eliminates checkpoint reusability, while the latter requires analysis that grows exponentially with speculation depth.

Effects of wrong-path instructions. Although the effects of wrong-path instructions on the commit instruction stream are generally small [Cain et al. 2002], they cannot be ignored given our tight bias goals. Errors in wrong-path modeling cause the schedule of wrong-path execution to differ from a simulation where all state is available, which in turn perturbs the execution schedule of the commit instruction stream.

We measure the bias introduced if we restrict live state to contain only state accessed by correct path instructions. With restricted live state, we omit all architectural state (memory values) and microarchitectural state (cache tags and branch predictor entries) that are not accessed in the simulation window during live point creation, leaving this state uninitialized (effectively random). A live point with restricted live state contains the smallest possible subset of state that can still simulate correct-path instructions (but will not accurately simulate wrong-path). Although the average bias increase for CPI is only 0.1%, the worst case is 3.3%. Figure 4.4 shows the bias results for the benchmarks with the most error.

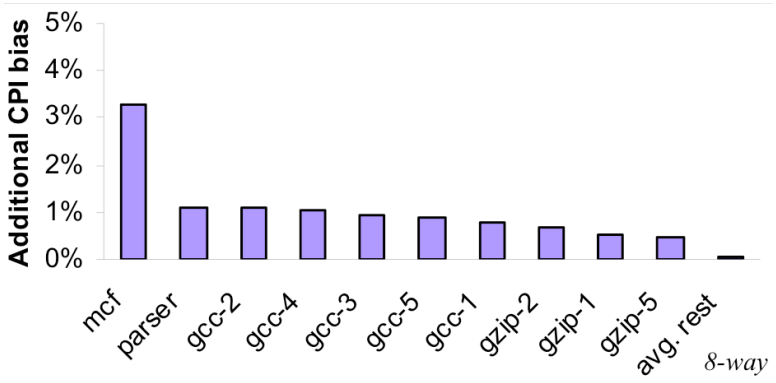


Figure 4.4: Restricted live-state bias

If only correct-path state is stored, wrong-path instructions are not accurately simulated.

Wrong-path instructions interact with the commit stream through resource contention and in the cache tag arrays. In the vast majority of cases, we can use branch predictor outcomes to identify the wrong-path instruction sequence, and cache tag arrays to identify wrong-path load latency. This information is sufficient to identify contention and cache tag array updates arising from speculative execution, without the need for the values accessed by wrong-path loads.

In our live state approach, we include the microarchitectural state necessary to reflect wrong-path effects (branch predictor, cache tag arrays, TLBs), but omit memory values unless they are accessed on the correct-path. By omitting the vast majority of memory values, the live state approach reduces storage requirements from over 100 MB to 142 KB per live point (uncompressed; assuming cache hierarchy and branch predictor of our 8-way baseline). Under this approach, unavailable memory values enter the microarchitecture (via a wrong-path load) on average less frequently than once per detailed window. We measured no appreciable increase, < 0.1% difference, in CPI bias over full warming.

4.2 Framework

One of the benefits of the live point design is that each live point is independent of all other live points, and can thus be processed in isolation. As others have noted [Girbal et al. 2003; Lafage and Seznec 2000; Lauterbach 1994], window independence allows a simulation to be parallelized across hosts (with parallelism degree up to the sample size). However, we can also leverage live point independence to minimize the runtime of absolute and comparative experiments, and provide results from simulations that are still in progress. The following subsections present a sampling methodology for absolute and comparative performance studies.

4.2.1 Absolute performance estimates

To report meaningful estimated results, a sampled simulation must complete processing of an unbiased sample of the complete benchmark. With functional warming, where the measurements must be processed in strict program order, the measured sample represents the entire benchmark only after the entire simulation is complete.

With independent live points, we are not forced to process detailed windows in program order. We can exploit this property to rearrange the live point processing order so that we can report unbiased performance estimates (with lower statistical confidence than final results) at any time.

A complete live point library forms an unbiased random (or systematic) sample of a benchmark. If we select a random sub-sample from the live point library, we arrive at a smaller, but still unbiased, random sample of the benchmark. Based on this principle, if we shuffle a live point library into random order, after each live point is simulated, the live points processed thus far form an unbiased random sample of the benchmark.

We exploit random-order live point processing to allow a simulation to report results at any time. As live points are processed, we calculate the confidence achieved in the sample observed thus far. As the sample size grows, the confidence improves, and the estimated results converge to their true values. As soon as we are satisfied with the current confidence, we can terminate the simulation. We impose a minimum sample size of 30 live points to ensure that the central limit theorem holds and our confidence calculations are valid [Jain 2001].

Online monitoring of simulation results and their current confidence has proven valuable during simulator development to get quick-and-dirty performance estimates and

detect simulator bugs. Even after processing a small sample (100's of live points), confidence intervals will be tight enough to identify gross performance bugs reliably.

To maximize simulation processing speed, we recommend shuffling live points on disk, prior to simulation. Live points should be stored in a single compressed file to maximize I/O performance (which is the performance-limiting bottleneck in our environment).

4.2.2 Comparative performance estimates

When a live point library is created, we set an upper bound on the sample size that can be measured with that library (i.e., the number of live points in the library). The upper bound is typically based on the sample size required to meet a desired statistical confidence for a benchmark and baseline microarchitecture combination. Because the required sample size will increase when a new microarchitecture has higher target metric variability (e.g., CPI variance), a live point library sized for the baseline configuration may fall short of the sample required for an experimental case.

In such comparative studies, researchers are often more interested in the relative performance of two designs than absolute performance. We can take advantage of this observation through a sampling procedure called matched-pair comparison, first proposed for computer architecture simulation sampling by Ekman and Stenström [2005]. Matched-pair comparison exploits the phenomenon that the change in performance from design x to design y tends to vary less than the absolute performance of either design. As a result, the change in performance can be assessed to a given confidence with a smaller sample than absolute performance.

Under matched-pair comparison, we build a confidence interval directly on the change in performance. Unlike an unpaired comparison of two different samples, in matched-pair

comparison, we measure the same sample (i.e., same live points) in each of two designs and compute the performance delta on each measurement interval. In the common case, the design change has a similar effect in all measurement intervals (e.g., a larger cache tends to improve performance uniformly by a small increment). Thus, the variance of the performance deltas, and required sample size, is small. The calculations and procedure for applying matched-pair comparison are detailed fully in [Ekman and Stenström 2005].

Ekman and Stenström report that matched-pair comparison typically reduces sample size by an order of magnitude compared to absolute performance estimates over a range of microarchitectural design changes. We performed a similar set of sensitivity studies (e.g., varying latencies, queue sizes, functional unit mix, etc.). Our results corroborate [Eckman and Stenström 2005], indicating that matched-pair comparison reduces sample size by a factor of 3.5 to 150. We note that matched-pair comparison is particularly effective for detecting that a design change has no appreciable impact (i.e., less than 3% CPI change). When a design change has little effect, nearly all measurement intervals behave identically under the base and experimental cases, resulting in low CPI-delta variance.

Matched-pair comparison addresses the risk that a comparative performance study will exhaust the available live point library without achieving the desired confidence. If we size a live point library such that it can achieve a particular confidence in an absolute estimate of the base case, we will typically require only a fraction of this library for comparative studies.

We can combine matched-pair comparison with random-order processing to report results online for comparative studies. The combined optimizations are particularly effective for rapidly searching a design space to eliminate designs that do not differ significantly from

the base case. A 50 measurement sample can rapidly distinguish design changes with no impact from those that require further simulation.

4.2.3 Experiment procedure

We now summarize our complete procedure for experimentation with live points. Figure 4.5 illustrates the steps in the procedure.

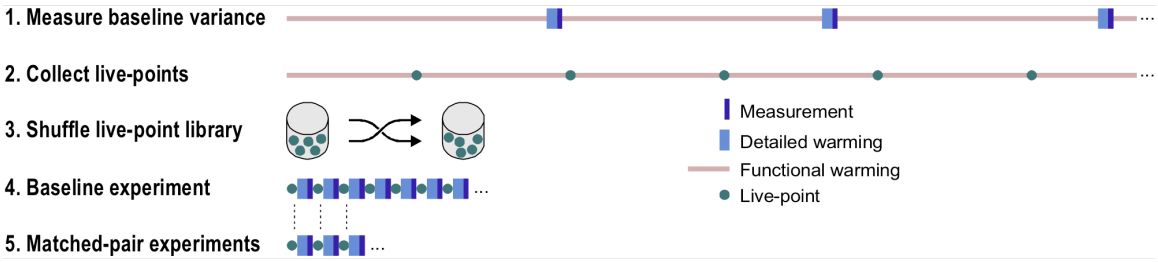


Figure 4.5: Live-point experimental procedure

Matched-pair experiments produce estimates of performance deltas from the baseline.

First, we must measure the target metric variance for the baseline configuration to determine an appropriate live point library size. We can measure variance using prior simulation sampling approaches, or estimate it from published results (Section 3.3). In our implementation, these simulations require seven hours on average for SPEC CPU2000.

Second, we must generate a live point library. We choose the maximum cache hierarchy and set of branch predictors of interest, and run a full-warming simulation that outputs compressed live points. Live point generation requires on average 8.5 hours per benchmark.

Third, we shuffle these live points into a random order and store them in a single compressed stream. Optionally, the live point library can be split into multiple compressed streams for parallel processing. Shuffling is compression-speed bound, and requires several minutes per benchmark.

Fourth, we measure the baseline configuration with our live point library. We record metrics of interest (e.g., CPI) for each live point. This simulation can be parallelized and can employ the random-order processing optimization. For our 8-way microarchitecture, this simulation reaches 99.7% confidence of ±3% error in an average of 91 seconds per benchmark (without parallelization).

Finally, we can perform comparative studies relative to the baseline microarchitecture using the live point library. These simulations can employ parallelization, random-order processing, and matched-pair comparison optimizations. Furthermore, we can monitor simulation results online, and terminate simulations at any time to report results with reduced confidence. If we assess our 16-way microarchitecture relative to our 8-way baseline, the simulation reaches target confidence in an average of 2.4 minutes per benchmark, while an absolute measurement of the 16-way microarchitecture requires 7.6 minutes per benchmark.

4.3 Results

In this section, we report results on the effectiveness of the live state approach in reducing storage cost and compare the performance of live points to other simulation sampling approaches.

4.3.1 Live state results

The live state approach is highly effective at reducing the storage cost of live points. Because the simulation window covered by each live point is short (a few thousand instructions), only ~16 KiB of memory state must be stored.

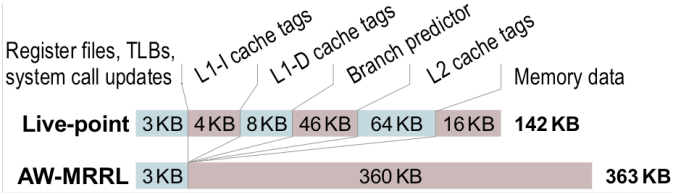


Figure 4.6: Breakdown of a typical live-point (uncompressed)

For comparison, a conventional checkpoint is 105 MiB on average.

Live state can also be used in conjunction with adaptive warming. However, because the simulation window required for cache warming is large (on average 4.1 million instructions per window), the required memory state is much larger, on average 360 KiB. Figure 4.6 compares the uncompressed size of live points (assumes cache/branch predictor of the 8-way microarchitecture) and live state for adaptive warming using MRRL (AW-MRRL; microarchitecture independent). We typically obtain 5:1 compression with gzip.

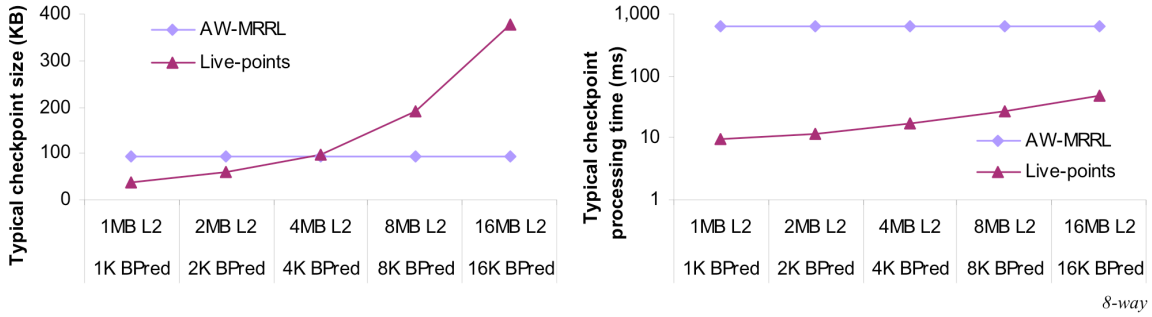


Figure 4.7: Compressed checkpoint size and processing time

Live points have a size advantage until large cache tag arrays are required. However, even for large caches, live points are much faster than adaptive warming using MRRL because no functional warming is needed.

The storage cost (and thus decompression/load time) of live points grows as the size of the stored microarchitectural structures increases. With adaptive warming, no microarchitecture-specific state is stored, and thus storage cost is fixed. As a result, there is a break-even point where the storage cost of live points and adaptive warming become equal. Figure 4.7 (left) shows that this break-even threshold occurs around a 4 MiB maximum cache size. However, for microarchitecture state larger than this threshold, live points remain an order of magnitude faster (Figure 4.7 right) because generating cache

of magnitude faster (Figure 4.7 right) because generating cache state dynamically is much slower than loading it from disk.

4.3.2 Live points performance

We use live points to estimate the absolute CPI of our benchmark suite to the same accuracy and confidence as previous simulation sampling techniques as described in Section 4.1.1. Table 4.1 presents measured run-time results for live points. Runtime results were collected with serial live point processing and only a single simulation running per system. We compare live points to non-sampled runs of the complete benchmark with SimpleScalar's sim-outorder, full warming using SMARTSim, and adaptive warming using MRRL (AW-MRRL). We show the best, average, and worst runtimes for the two microarchitectural configurations introduced in Section 4.1.1.

Table 4.1: Live-points runtimes compared to SMARTS and adaptive warming

We include the fastest and slowest runtimes of SPEK2K benchmarks to show the variability of each technique.

	8-way (1MB L2)					16-way (4MB L2)				
	Minimum		Average	Maximum		Minimum		Average	Maximum	
sim-outorder	2.2 h *perlbmk*	13 h *gcc-2*	**5.5 d**	15 d *mgrid*	24 d *parser*	3.8 h *perlbmk*	22 h *gcc-2*	**9.6 d**	27 d *mgrid*	42 d *parser*
SMARTSim	4.4 m *perlbmk*	29 m *gcc-2*	**7.0 h**	17 h *mgrid*	25 h *parser*	4.6 m *perlbmk*	31 m *gcc-2*	**7.3 h**	18 h *mgrid*	26 h *parser*
AW-MRRL	61 s *perlbmk*	88 s *eon-2*	**1.5 h**	7.1 h *ammp*	9.5 h *parser*	65 s *perlbmk*	92 s *eon-2*	**1.6 h**	7.5 h *ammp*	9.9 h *parser*
Live-points	1 s *swim*	2 s *eon-2*	**91 s**	5.0 m *vpr*	12 m *ammp*	13 s *swim*	14 s *eon-2*	**7.6 m**	25 m *vpr*	1.3 h *ammp*

Times are specified in days (d), hours (h), minutes (m), or seconds (s).

Live points eliminate the functional warming bottleneck in SMARTSim, reducing average simulation time for SPEC CPU2000 benchmarks from 7 hours to just 1.5 minutes (8-way baseline microarchitecture). Live points are 50 times faster than AW-MRRL. Live point simulations often complete faster than native execution of benchmarks on our host platform, which typically requires several minutes per benchmark.

For both SMARTSim and sim-outorder, simulation time varies linearly with benchmark length. Thus, we can expect simulation times to grow with longer benchmarks. In contrast, runtime with live points and AW-MRRL depends on sample size, and thus CPI variability. We do not observe any relationship between CPI variability and benchmark length; therefore, we do not expect live points' runtimes to increase for longer benchmarks.

Table 4.2: Summary of simulation sampling warming methods

	Complete Simulation (sim-outorder)	**Full Warming (SMARTS)**	**Adaptive Warming (AW-MRRL)**	**Checkpointed Warming (Live-points)**
Average (worst) CPI bias	None	0.6% (1.6%)	1.1% (5.4%)*	0.6% (1.6%)
Average benchmark runtime	5.5 days	7.0 hours	1.5 hours	91 seconds
Scaling behavior	O($B \times DS$)	O(B)	O(1)	O(C)
Independent checkpoints	N/A	N/A	No*	Yes
SPEC2K checkpoint library size	N/A	N/A	30 GB	12 GB (1 MB L2)
Scaling behavior	N/A	N/A	O(1)	O(C)
Fixed microarchitecture parameters	None	None	None	Max cache, TLB, branch predictors

B = benchmark length, C = max cache size, DS = detailed simulation speed
*AW-MRRL can produce independent checkpoints, but bias increases to 1.9% average, 11% worst.

Table 4.2 summarizes the characteristics of the warming approaches evaluated in this paper. The table shows the live point library sizes, run times, and biases measured for each technique.

Live points match the bias of SMARTSim. AW-MRRL with a reuse distance threshold of 99.9% does not match this tight error. Adaptive warming accuracy may improve with a higher reuse threshold, at the cost of further slowdown relative to live points. Sampling error can be made arbitrarily small with all three warming approaches by increasing sample size.

Table 4.2 also indicates the scaling behavior of live point size and processing time with respect to microarchitectural model and benchmark characteristics, and indicates what microarchitecture model parameters must be fixed when live points are created. A live

point library restricts maximum cache and TLB sizes and must include state for each branch predictor used in subsequent simulations. However, other microarchitectural configuration parameters are not fixed. Live points are independent of one-another, enabling parallel simulation and online results reporting.

4.4 Related work

Van Biesbrouck, Eeckhout, and Calder [2005] apply a checkpointed warming approach similar to live points to accelerate SimPoint measurement. They report that checkpoint libraries for SimPoint-derived samples typically require less storage than high-confidence (i.e., 99.7% confidence of ±3% error) uniform samples, whereas uniform samples simulate fewer instructions in detail per benchmark (~30 million rather than ~300 million instructions) and result in shorter simulation turnaround. Our experiments corroborate these results. However, with uniform sampling, we can trade off confidence in results to reduce turnaround time and live point storage cost. Existing representative sampling techniques do not provide quantitative measures of confidence with each result. Moreover, online result reporting is not applicable to representative sampling.

Wenisch et al. [2006] describes our extension of the SMARTS sample design to a critical class of multiprocessor server workload. We provide a new sampling population definition for *throughput applications*—the server side of client-server applications, such as the Transaction Processing Performance Council (TPC) database and SPECweb workloads. We leveraged the random nature of transaction arrivals in these applications to construct a meaningful random sample despite deterministic simulation models. Furthermore, to obtain tractable samples for these applications, we measured and validated fine-grain progress metrics that are proportional to transaction completion rates. We used the full-

system multiprocessor simulator, Flexus [Hardavellas et al. 2004], and our multiprocessor checkpoint implementation. Checkpoint sampling enable a multiprocessor simulation turnaround of only 10 to 100 CPU hours rather than the 10 to 20 CPU years required without sampling.

5 Stratified sampling

Two different approaches to sample selection are: (1) statistical uniform sampling of a benchmark's instruction stream, and (2) targeted sampling of non-repetitive benchmark behaviors. Uniform sampling, such as the SMARTS framework, has the advantage that it requires no foreknowledge or analysis of benchmark applications, and it provides a statistical measure of the reliability of each experimental result. However, this approach ignores the vast amount of repetition within most benchmark's instruction streams, taking many redundant measurements. Targeted sampling instead categorizes program behaviors to select fewer measurements, reducing redundant measurements. The SimPoint approach [Sherwood, Perelman, and Calder 2002] identifies repetitive behaviors by summarizing fixed-size regions of the dynamic instruction stream as *basic block vectors (BBV)*, building clusters of regions with similar vectors, and taking one measurement within each cluster.

Benefits of both sampling approaches, statistical confidence and reduced measurement, can be achieved by placing the phase identification techniques of targeted sampling in a statistical framework that provides a confidence estimate with each experiment. *Stratified random sampling* is this statistical framework. Stratified sampling breaks a population into strata, analogous to targeted sampling, and then randomly samples within each stratum, as in uniform sampling. By separating the distinct behaviors of a benchmark into different strata, each behavior can be characterized by a small number of measurements. Each of these characterizations is then weighted by the size of the stratum to compute an overall estimate. The aggregate number of measurements can be lower than the number required by uniform sampling.

The effectiveness of stratified sampling can be evaluated along two dimensions. First, it might reduce the total quantity of measurements required. For simulators where a large number of measurements implies significant cost—for example, the storage of large architectural state checkpoints to launch each measurement—a reduction of measurements would imply cost savings.

More commonly, the total number of instructions measured has the larger impact on simulation cost. To improve total measurement, a stratification approach must reduce the quantity of required measurements while maintaining the small measurement sizes achievable with simple random sampling.

We evaluate the practical merit of combining sample targeting with statistical sampling in the form of stratified random sampling. We perform an oracle limit study to establish bounds on improvement from stratification and evaluate two practical stratification approaches: program phase detection and IPC profiling. We evaluate both approaches quantitatively in terms of *sample size* (measurement quantity) and *sampling unit size* (measurement size), and qualitatively in terms of the upfront cost of creating a stratification. We demonstrate:

1. **Limited gains in sample size:** We show that stratifying via program phase detection achieves only a small reduction in sample size over uniform sampling, 2.2 times, in comparison to the oracle opportunity of 43 times. Phase detection assures that each stratum has a homogenous instruction footprint. Unfortunately, data effects and other sources of performance variation remain. The reduction in CPI variability achieved by stratifying on instruction footprint is not sufficient to approach the full opportunity of stratification.

2. **Expensive analysis and limited applicability:** We show that IPC profiling requires an expensive analysis that is microarchitecture specific, and its gains do not justify this cost.
3. **No improvement in total measurement:** We show that neither stratification approach improves over simple random sampling in terms of total instructions measured. Because of the computational complexity of clustering, neither stratification approach can be applied at the lowest sampling unit sizes achievable with random sampling. This increase in sampling unit size offsets reductions in sample size for stratified sampling.

The remainder of this chapter is organized as follows. Section 5.1 presents stratified random sampling theory and details how to correctly achieve confidence in results from a stratified population. Section 5.2 discusses our optimal stratification study, while Section 5.3 covers our evaluations of two practical stratification techniques. Both sections cover the improvements to sample size and total measured instructions as compared to simple random sampling for each technique.

5.1 Framework

The confidence in results of a simple random sample is directly proportional to the sample size and the variance of the property being measured. The sample size is the number of measurements taken to make up a sample, and variance is the square of standard deviation. Significant reductions in sample size can often be achieved when a population can be split into segments of lower variance than the whole.

Stratified random sampling of a population is performed by taking simple random samples of *strata*, mutually exclusive segments of the population, and aggregating the resulting

estimates to produce estimates applicable to the entire population. Strata do not need to consist of contiguous segments of the population, rather every population member is independently assigned to a stratum by some selection criteria. If stratifying the population results in strata with relatively low variance, a small sample can measure each stratum to a desired confidence. By combining the measurements of individual strata, we can compute an overall estimate and confidence. With low variance strata, the aggregate size of a stratified sample can be much smaller than a simple random sample with equivalent confidence. A population whose distinct behaviors are assigned to separate strata will see the largest decreases in sample size when using stratified sampling.

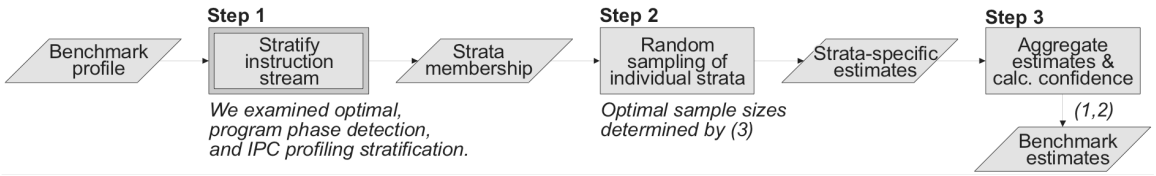


Figure 5.1: Stratified random sampling process

We focus on the relative effectiveness of two practical stratification approaches for step 1 in this work. The referenced equations for steps 2 & 3 are in Section 5.1.

The process of stratified random sampling is illustrated in Figure 5.1. The first of three steps is to stratify the population into K strata. We discuss various techniques for stratifying populations in the context of microarchitecture simulation in Sections 5.2 and 5.3. Second, we collect a simple random sample of each stratum. We represent the variable of interest as x, and strata-specific variables with the subscript h, where h ranges from 1 to K. Therefore, N_h is the population size of stratum h, n_h is the sample size for stratum h, while σ_{hx} is that stratum's standard deviation of x. The final step is to aggregate the individual stratum estimates to produce estimates of the entire population. A simple weighted mean is used to produce a population mean estimate:

$$\bar{x} = \frac{\sum (N_h / N)\bar{x}_h}{K}$$

Equation 1: Estimated stratified population mean

where the summation is over all strata of the population (h = 1 to K); thus, $\sum N_h = N$, and $\sum n_h = n$. Note, we assume $N_h \gg n_h \gg 1$ to simplify the stratified sampling expressions. The confidence interval of a mean estimate from a stratified random sample is determined by:

$$\pm(\varepsilon \cdot \bar{X}) \approx \pm z \sqrt{\sum \left(\frac{N_h}{N}\right)\left(\frac{\hat{\sigma}_{hx}^2}{n_h}\right)}$$

Equation 2: Confidence interval of estimated stratified population mean

where z is the 100[1 – (α / 2)] percentile of the standard normal distribution (z = 2.0 for 95% and z = 3.0 for 99.7% confidence). Note that a sampling estimate of a stratum's standard deviation is marked with a hat as $\hat{\sigma}_{hx}$.

The required sample size for each stratum, n_h, which produces a desired overall confidence interval with minimum total sample size n can be calculated if the standard deviation of each stratum σ_{hx} is known or can be estimated. The procedure for calculating the optimal stratified sample is known as *optimal sample allocation* [Levy and Lemeshow 1999]. To determine an optimally-allocated stratified sample for a desired confidence interval we first calculate the total stratified sample size:

$$n \geq \frac{\left(\frac{z^2}{N^2}\right)\left(\sum \frac{N_h^2 \sigma_{hx}^2}{\pi_h \bar{X}^2}\right)}{\varepsilon + \left(\frac{z^2}{N^2}\right)\left(\sum \frac{N_h \sigma_{hx}^2}{\bar{X}^2}\right)} \text{ where } \pi_h = \frac{N_h \sigma_{hx}}{\sum N_h \sigma_{hx}}$$

Equation 3: Optimal stratified sample size

The sample size of each stratum is the fraction π_h of the total stratified sample size n; individual stratum sample sizes are $n_h = \pi_h \cdot n$.

5.2 Optimal stratification

In order to evaluate practical stratification approaches for the experimental procedure presented in Section 5.3, we first quantify the upper bound reduction in sample size achievable with an optimal stratification. We focus on CPI as the target metric for our estimation, and use the same 8-way and 16-way out-of-order superscalar processor configurations, SPEC CPU2000 benchmarks, and simulator codebase described in Section 3.1.2.

Determining an optimal stratification for CPI requires knowledge of the CPI distribution for the full length of an application—knowledge which obviates the need to estimate CPI via sampling. To perform this study, we have recorded complete traces of the per-unit IPC (not CPI, for reasons explained later) of every benchmark on both configurations. While not a practically applicable technique, this study establishes the bounds within which all practical stratification methods will fall. At worst, an arbitrary stratification approach will match simple random sampling, as random assignment of sampling units to strata is equivalent to simple random sampling. At best, any approach will match the bound established here.

Optimal stratified sampling: To minimize total sample size, we need to determine an optimal number of strata, and minimize their respective variances. Then, we calculate the correct sample size for a desired confidence using the optimal stratified sample allocation Equation 3. This equation provides the best sample size for each stratum, given their variances and relative sizes. Larger and higher variance strata receive proportionally larger samples. We constrain sample size for each stratum to a minimum of 30 (or the entire stratum, if it contains fewer than 30 elements) to ensure that the central limit theorem holds, and that our confidence calculations are valid [Levy and Lemeshow 1999].

The optimal number of strata, K, cannot be determined in closed form. Intuitively, more strata allows finer classification of application behavior, reducing variance within each stratum, and therefore reducing sample size. However, at some critical K, the floor of 30 measurements per stratum dominates and increasing K increases sample size. For each combination of benchmark, microarchitecture, and sampling unit size, U, we determine the total stratified sample size for each value of K up to the optimal value, by starting with $K = 1$ and stopping when total sample size decreases to a minimum.

For each value of K, we determine the optimal assignment of sampling units to strata such that the CPI variance of each stratum is minimized. We employ the k-means clustering algorithm, using the implementation described in [Pelleg and Moor 1999] that utilizes *kd*-trees and blacklisting optimizations. The k-means algorithm is one of the fastest clustering algorithms, and the implementation in [Pelleg and Moor 1999] is optimized for clustering large data sets, up to approximately 1 million elements. (Beyond 1 million elements, the memory and computation requirements render the approach infeasible.) Each k-means clustering was performed with 50 random seeds to ensure an optimal clustering result. To stratify the large populations of SPEC CPU2000 benchmarks at small U (on average 174 million sampling units per benchmark at $U = 1000$ instructions), we must reduce the data set before clustering. Figure 5.2 illustrates how we reduce the data set without impacting clustering results. We assign sampling units to bins of size 0.001 IPC, and then cluster the bins using their center and membership count. We bin based on IPC rather than CPI as IPC varies over a finite range for a particular microarchitecture (i.e., 0 to 8 for our 8-way configuration, thus, 8000 bins). As long as the number of bins is much larger than K, and the variance within a bin is negligible relative to overall variance, binning does not adversely affect the results of the clustering algorithm.

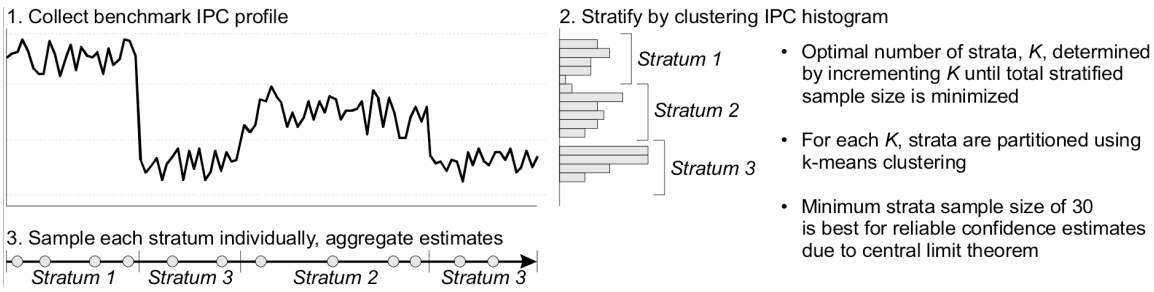


Figure 5.2: Optimal stratification for a particular benchmark and microarchitecture

Collecting the IPC profile requires performance simulation of the full length of the target benchmark.

After each clustering, we calculate the variance of the resulting strata and determine an optimal sample size as previously described. We iterate until the critical value for K is encountered. The optimal K lies between one and ten clusters for all benchmarks and configurations that we studied, and tends to decrease slightly with increasing U. Note that the optimal K is independent of the target confidence interval.

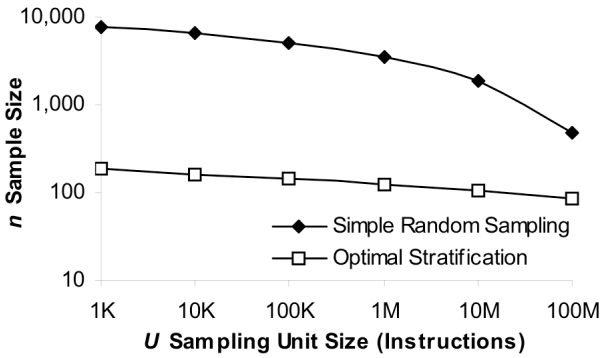


Figure 5.3: Optimal stratification's mean sample size vs. simple random sampling

Mean sample size per benchmark for SPEC CPU2000 with the 8-way processor configuration.

Impact on sample size: Figure 5.3 illustrates the impact of stratification on sample size, n, for the 8-way configuration. The top line in the figure represents the average sample size required for a simple random sample to achieve 99.7% confidence of ±3% error across all benchmarks. The bottom line depicts the average sample size with optimal stratification. Stratification can provide a 43 times improvement in sample size for U = 1000 instructions, reducing average sample size from ~8000 to 185 measurements per benchmark. This result

demonstrates that random sampling takes many redundant measurements, and that there is significant opportunity for improvement with an effective stratification technique.

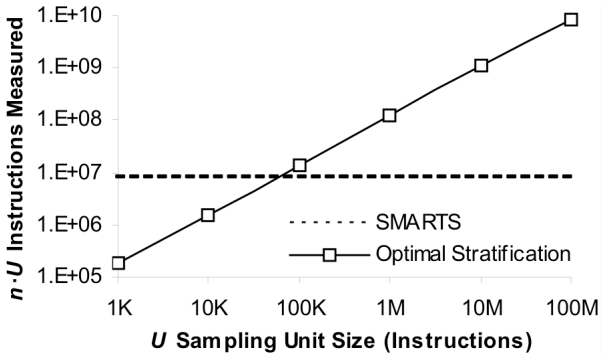


Figure 5.4: Total measured instructions per benchmark with optimal stratification

Impact on total measured instructions: Figure 5.4 illustrates the impact of stratification on total measured instructions, $n \cdot U$. The dashed line illustrates the total instructions required for the SMARTS technique, which performs systematic sampling at $U = 1000$ instructions. The graph shows that any practical stratification approach must be applied at a unit size of 10,000 instructions or smaller in order to have a possibility of outperforming existing sampling methodology.

5.3 Results

The optimal stratification study presented in Section 5.2 establishes upper and lower bounds by which we can measure the effectiveness of any stratification approach. However, creating the optimal stratification requires knowledge of the CPI distribution for the full length of an application, and is optimal only for that specific microarchitecture configuration. In order for stratification to be useful, we must balance the cost of producing a stratification with the time saved relative to simple random sampling over the set of experiments which can use the stratification. Thus, we desire stratifications that can be

computed cheaply and can be applied across a wide range of microarchitecture configurations. In the following subsections, we analyze three stratification approaches.

5.3.1 SimPoint program phase detection

SimPoint [Sherwood, Perelman, and Hamerly 2002] presents program phase detection as a promising approach for identifying and exploiting repetitive behavior in benchmarks to enable acceleration of microarchitecture simulation. SimPoint identifies program phases based upon a basic-block vector profile. SimPoint clusters measurement units based on the similarity of portions of the BBV profiles.

Statistically Valid SimPoint [Perelman, Hamerly, and Calder 2003] presents a method for evaluating the statistical confidence of SimPoint simulations where only a single unit is measured from each cluster. However, the proposed use of parametric boot-strapping only provides confidence interval estimates for the specific microarchitecture where the boot-strap is performed, and does not account for individual experiment's variations in performance. In addition, this analysis requires CPI data for many points within each cluster.

Instead, by applying BBV phase detection in the context of stratified random sampling, we can obtain a confidence estimate with every experiment. By measuring at least 30 units from each stratum (BBV cluster), we satisfy the conditions of the central limit theorem and obtain a confidence estimate with each simulation experiment. The number of strata was optimally selected using the same technique as our optimal stratification study in Section 5.2. SimPoint seems a promising approach for stratification, as it achieves both of the goals outlined earlier. First, basic block vector analysis is relatively low cost, as it can be accomplished using a BBV trace obtained by functional simulation or native execution of

instrumented binaries (if experimenting with an implemented ISA). Second, basic block vectors are independent of microarchitecture, and thus, the resulting stratification can be applied across many experiments.

Practical costs: The primary costs of program phase stratification are the collection of a benchmark's raw BBV data and the clustering analysis time. Collection of BBV data can be done with native execution for existing instruction set architectures, otherwise functional simulation is required. For the unit sizes advocated in [Perelman, Hamerly, and Calder 2003] and [Sherwood, Perelman, and Hamerly 2002] of 1 million to 100 million instructions, analysis time for clustering is a few hours at most. However, clustering quickly becomes intractable as we reduce U further. It is infeasible to compute a k-means clustering for $U < 100{,}000$ instructions, since, for most SPEC CPU2000 benchmarks, this results in more than 1 million sampling units. The high dimensionality (15 dimensions after random linear projection) of BBV data prevents the binning optimization done for the optimal stratification study in Section 5.2 due to the sparseness of the vector space.

Impact on sample size: Program phase detection does provide a modest improvement in sample size over simple random sampling. However, phase detection falls short of optimal stratification since it seeks to ensure the homogeneity of the instruction footprint of each stratum. This does not necessarily lead to minimal CPI variance within each stratum. On average, program phase clustering improves sample size by only 2.2 times over simple random sampling as shown in Figure 5.5. The average sample size at $U = 1$ million instructions was 3590 for simple random sampling and 1615 for BBV stratified random sampling, as compared to 125 for optimal stratified sampling.

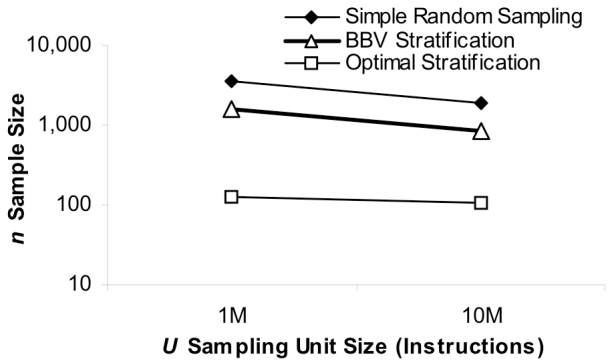


Figure 5.5: BBV program phase stratification mean sample size

BBV stratification reduces average sample size by 2.2 times over simple random sampling, but it requires $U > 100{,}000$.

Impact on total measured instructions: Because the BBV clustering analysis cannot be performed for U below 100,000, stratification based on program phase cannot match the total measured instructions achievable with simple random sampling. With $U = 1$ million, BBV stratification results in an average of 1.6 billion instructions measured per benchmark, while a simple random sample with $U = 1000$ requires only 8 million instructions per benchmark to be measured.

5.3.2 Dynamic program phase detection

Dynamic Sampling [Falcón, Faraboschi, and Ortega 2007] presents an alternative phase detection approach to SimPoint. Phases, and corresponding measurement locations, are dynamically chosen during the simulation of the benchmark application instead of prior to the simulation. Execution statistics are collected during simulation for fixed-sized instruction windows. When the rate of a selected execution event changes by more than a threshold percentage between windows, a new phase is considered to have begun. Warming and the measurement of the following pair of fixed-size window is performed, and fast forwarding is then resumed.

Several execution statistics, and threshold values were studied by Falcón et al., but the best performing and recommended configuration used the miss rate of the simulator's code translation cache to detect phase changes. The fixed-size execution windows were set to 1 million instructions each, and a 300% change in miss rate between windows indicated a new phase. Upon detecting the phase change, the detailed simulator performed 1 million instructions warming, then 1 million instructions measurement.

Our analysis of their runtime results indicate that for SPEC CPU2000, each benchmark had approximately 1,000 phase changes on average, and therefore, approximately 1,000 measurements of 1 million instructions each. Falcón et al. evaluated their dynamic sampling framework on an x86 instruction set simulation of an out-of-order superscalar microarchitecture. Their results show an average error of 1.1%, as compared to 0.5% for SMARTS, and 1.7% for SimPoint. Their advantage over SMARTS lies in their speedup due to their use of native execution to fast forward between detected phases, as opposed to functional warming. Their implementation of SMARTS obtained a 7.4 times speedup versus a full detailed simulation, versus 158 times speedup for dynamic sampling. As a point of comparison, Chen [2004] implemented and evaluates SMARTS with native execution support of functional warming achieving a 96 times speedup.

Given that dynamic sampling uses a sample size of 1,000 that is as large as the recommended sample size for simple random sampling at U = 1 million instruction (derived from the V_{CPI} illustrated in Figure 3.2), we expect accurate performance estimates from dynamic sampling. We investigated how much of their accuracy is due to their large sample size, and how much accuracy was gained by using their dynamic sampling phase detection versus systematic sampling. We performed our comparison using the 8-way out-of-order supersca-

lar processor configuration, SPEC CPU2000 benchmarks, and simulator codebase described in Section 3.1.2.

Practical costs: We configured a systematic sampling experiment to have the same cost as dynamic sampling in order to compare the two approaches with an equal number of measured instructions. In addition, we did not use functional warming, but rather we used the same 1 million instructions of detailed warming for both dynamic sampling and systematic sampling. We configured SMARTSim to track the miss rate of an equivalent structure to a code translation cache (a supplementary 256 MiB instruction cache) to determine phase changes for dynamic sampling. Both sampling approaches has measurement units of 1 million instructions, and both were calibrated to take an average of 1,000 measurements per benchmark (1 billion instructions total per benchmark).

Table 5.1: Performance comparison between dynamic and systematic sampling

Configuration	Mean IPC error	Can use native execution
Systematic sampling with functional warming (SMARTS)	0.66%	no
Systematic sampling	1.9%	yes [Chen 2004]
Dynamic sampling	2.1%	yes [Falcón et al. 2007]

Table 5.1 lists the results of our comparison between dynamic sampling and systematic sampling. Both approaches achieve relatively low error, but systematic sampling does not rely on a heuristically chosen phase detection approach. Both approaches can use native execution, and will achieve the exact same speedups over full detailed simulation. Our conclusion is that dynamic sampling does not provide an advantage over systematic sampling when the sample size is as large as 1,000.

Impact on sample size: To determine if the phase detection approach of dynamic sampling can provide an advantage over systematic sampling at under different constraints, we examined the effect of drastically reducing the sample size. For dynamic sampling we increased the miss rate threshold that determines the beginning of new execution phases to reduce the sample size. We matched dynamic sampling's sample sizes with systematic sampling by lengthening the periods of fast forwarding between measurements.

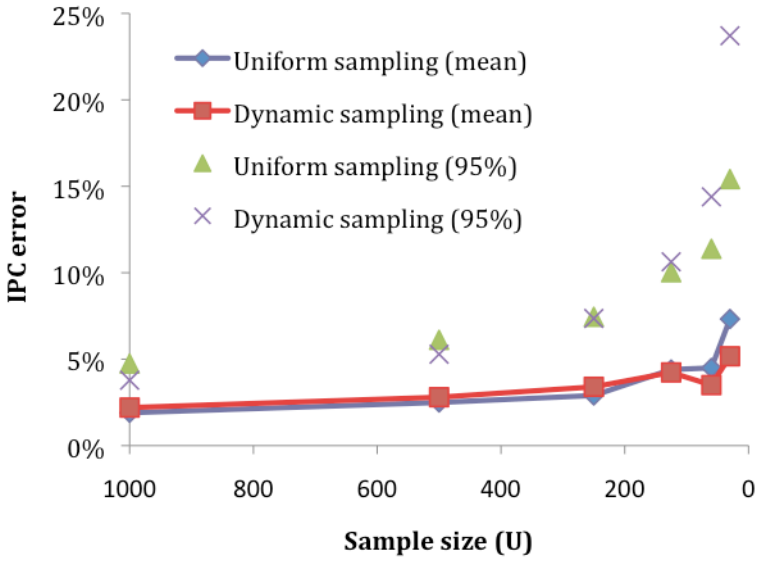


Figure 5.6: Increasing accuracy advantage of dynamic sampling

The accuracy improvements provided by dynamic sampling's phase detection algorithm become evident as smaller sample sizes are used. However, the error outliers for dynamic sampling are larger than uniform sampling at small sample sizes (the 95th percentile errors are shown among the SPEC CPU2000 benchmarks).

Figure 5.6 plots our results when we reduced the sample sizes of both systematic and dynamic sampling. The accuracy advantage of systematic sampling is lost as the sample size is greatly reduced, showing that dynamic sampling's phase detection approach is more effective than systematic or simple random sampling when the number of measurements is constrained. Systematic sampling does not perform as well with small sample sizes because it has a low probability of measuring all the different phase behaviors in a benchmark, while dynamic sampling attempts to target these behaviors.

5.3.3 IPC profiling

The optimal stratification study in Section 5.2 achieves large gains with stratification by stratifying directly on the target metric, in this case CPI. Optimal stratification can not be done for each experiment in practice because it requires the very same detailed simulation that we are trying to accelerate. However, if it were possible to perform this expensive stratification *once* per benchmark on a test microarchitecture, and then apply this stratification to many other microarchitecture configurations over many experiments, the long term savings might justify the one time cost. The key question is whether strata with minimal variance on one microarchitecture also have low variance on another microarchitecture. We evaluate the promise of this approach by computing a stratification using an IPC profile of our 8-way processor configuration and evaluating this stratification when applied to the 16-way configuration. The two microarchitectures differ in their fetch, issue and commit widths, functional units, memory ports, branch predictor and cache configurations, and cache latency (details in Section 3.1.2).

Practical costs. This approach needs a trace of the IPC of every block of U instructions, requiring a detailed simulation of the entirety of every benchmark. The longest SPEC CPU2000 benchmarks require up to a month to simulate in detail. We have successfully clustered sampling units for U = 10,000, but storage requirements and processing time prevent clustering at U = 1000 instructions. Unlike the optimal stratification experiment of Section 5.2, practical use of IPC profile stratification requires storing the strata assignment of every sampling unit to disk (to allow strata selection for a second experiment), and the storage needs becomes prohibitive at U = 1000.

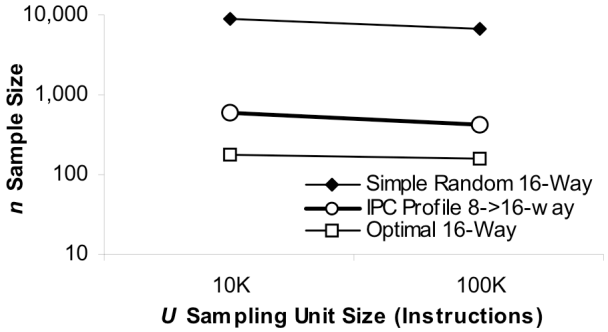


Figure 5.7: Mean sample size for IPC profile stratification

The 8-way profile's stratification was applied to the 16-way microarchitecture to determine if the stratifications were similar enough to produce an effective sampling result.

Impact on sample size. Two measurements units which have identical performance on one microarchitecture, and are thus members of the same stratum, may be affected differently by microarchitectural changes, increasing variance in the stratum. Thus, a larger sample is required to accurately assess the stratum. Figure 5.7 compares the sample size obtained with an 8-way IPC profile stratification to the optimum stratification and simple random sampling for the 16-way configuration. The 8-way stratification improves over purely random sampling by a factor of 15 times, as compared to an opportunity of 48 times for the 16-way microarchitecture. An IPC profile stratification will provide large returns only for microarchitectures very similar to the test microarchitecture that generated the profile.

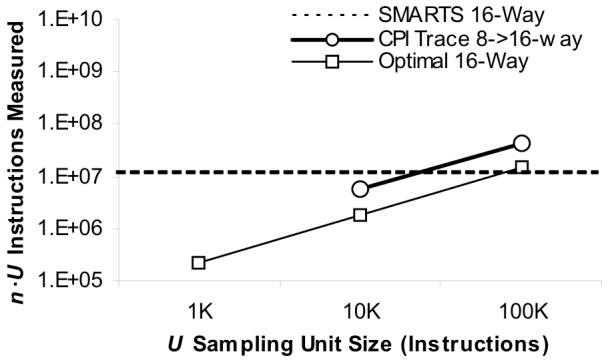


Figure 5.8: Total measured instructions per benchmark with IPC profile stratification

Impact on total measured instructions. As Figure 5.8 shows, IPC profile stratification at U = 10,000 roughly breaks even with SMARTS in terms of total measured instructions. This performance does not justify the significant one time cost of creating the stratification. Even if a method were developed which could stratify at U = 1000, the limited microarchitecture portability of the stratification renders it unlikely that the high cost of generating an IPC profile will be worthwhile.

6 High-resolution tracing

The “smart” features in modern microprocessors that enhance the runtime of typical applications actually increase the effort and expertise needed to hand-tune high performance applications. It is difficult for a programmer to fully anticipate dynamic effects: the overlapping interactions of caches, branch predictors, load/store units and queues, TLBs, prefetchers, and many unexposed mechanisms in the underlying hardware. Without such knowledge, seemingly sound code/algorithm optimizations to correct a supposed performance bottleneck can perform below expectations, or worse, have the opposite outcome.

Performance traces of program execution help software developers and computer architects improve their designs by identifying and explaining execution bottlenecks. In most computer systems, traces are collected by in-band performance counters, where the measured program’s performance is perturbed by the interruptions needed to collect a trace. The perturbations result in low-fidelity traces, where error increases with trace resolution (interruption frequency) until the true program performance is indiscernible. The perturbations introduced by the measurement process cannot be directly eliminated as long as system cost limitations prevent out-of-band performance probes and memory for performance trace data from being included in commodity computers.

We present two reconstruction algorithms that improve the resolution limit of high-fidelity performance traces. *Median reconstruction* improves the signal-to-noise ratio of repeated high-resolution traces of a program. *Super-resolution reconstruction* has a higher computational cost, but can combine staggered traces into an even higher-resolution trace. Both algorithms specifically target the skewed and heavy-tailed error distributions of CPU

microarchitectural performance measurements. Our goal is to obtain high-fidelity, high-resolution performance traces from commodity hardware using median and super-resolution reconstruction individually and in combination.

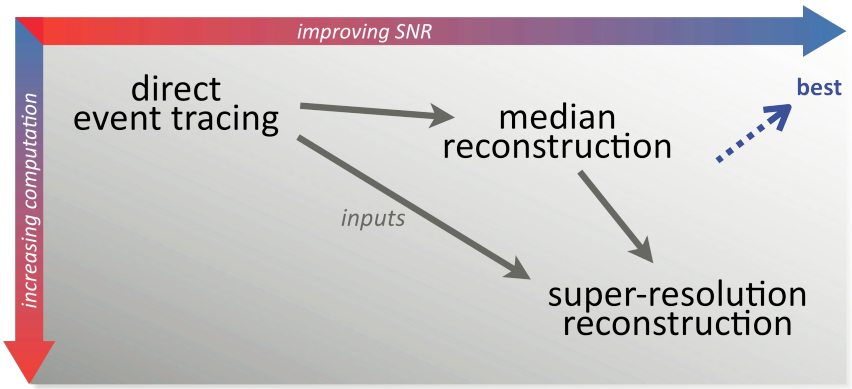


Figure 6.1: Performance tracing methods

Super-resolution trades error-tolerance for magnification.

Super-resolution reconstruction was inspired by existing image processing algorithms for super-resolution where multiple images of the same subject, with sub-pixel offsets, are used to reconstruct a higher-resolution resulting image. We attempt to collect and register multiple performance traces of a benchmark application using performance counters. These traces can then be used the conditions of a reconstruction problem that we solve to provide a higher-resolution trace.

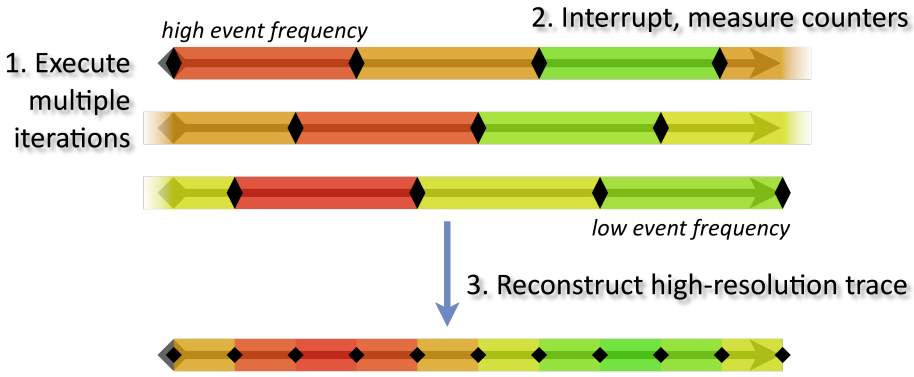


Figure 6.2: Super-resolution concept

Repeated executions of a benchmark application allow a high-resolution reconstruction of performance to pinpoint performance bottlenecks and use as input for further analysis.

We characterized the error observed on the an Intel Core 2 processor when executing the Intel Performance Primitive (IPP) kernels, and periodically interrupting to record the performance counter values to memory. The error of each individual measurement is defined as the event count deviation from the "ground truth" event count; the stable and unperturbed event count for each window of execution. The signal-to-noise ratio is used as a normalized metric to quantify and compare the fidelity of performance counter traces collected at various resolutions. Our objective is maximize resolution (i.e., minimize the size of each window of execution measured) while maintaining a SNR of 10 dB or greater. A SNR of 10 dB allows for accurate identification of program segments with exceptional performance and accurate characterization of performance bottlenecks.

We define the (directly-measured) "resolution limit" of a performance counter and executing application to be the maximum resolution that we can measure with a SNR of 10 dB for a given application. We called the approach of executing a program once, and periodically interrupting it to collect a performance trace, "direct measurement." Our two new algorithms for post-processing multiple sets of directly measured performance traces are called "median reconstruction" and "super-resolution reconstruction." Super-resolution reconstruction can use either direct measurements or median reconstructed measurements as input.

High-fidelity high-resolution performance traces can be used to empirically determine linear performance models of program segments. The linear models are solved using regression analysis, and can be used by programmers and architects to evaluate design tradeoffs.

6.1 Framework

This section presents background on the performance counters built into general purpose processors in Section 6.1.1, the sources of error that affect their accuracy in Section 6.1.2, the concept of super-resolution reconstruction in Section 6.1.3, and the notation and variables we use in Section 6.1.4.

6.1.1 Performance counters

General purpose, high performance, processors contain internal performance measuring hardware that is controllable via software. The x86, Power, IA-64 (Itanium), UltraSPARC, Cray, MIPS, and Alpha instruction set architectures all have vendor or model specific extensions that provide access to dynamic performance data. Most performance data is collected by simply counting execution events and signals (e.g., memory accesses, instruction retirements). Thus, the relevant components of performance measuring hardware are generally referred to as "performance counters."

Performance counters are usually comprised of three components: a counter register, event trigger sources, and configuration registers. The counter register is usually a 32–48 bit signed integer register with a dedicated adder circuit. Event trigger sources are microarchitectural components that are monitored by connecting wires from the component to the bank of counter registers. Configuration registers are associated with the event counters to control their behavior. Figure 6.3 illustrates the connections between these three components within a processor core. Most processors have from 2 to 4 performance counters that can be used simultaneously.

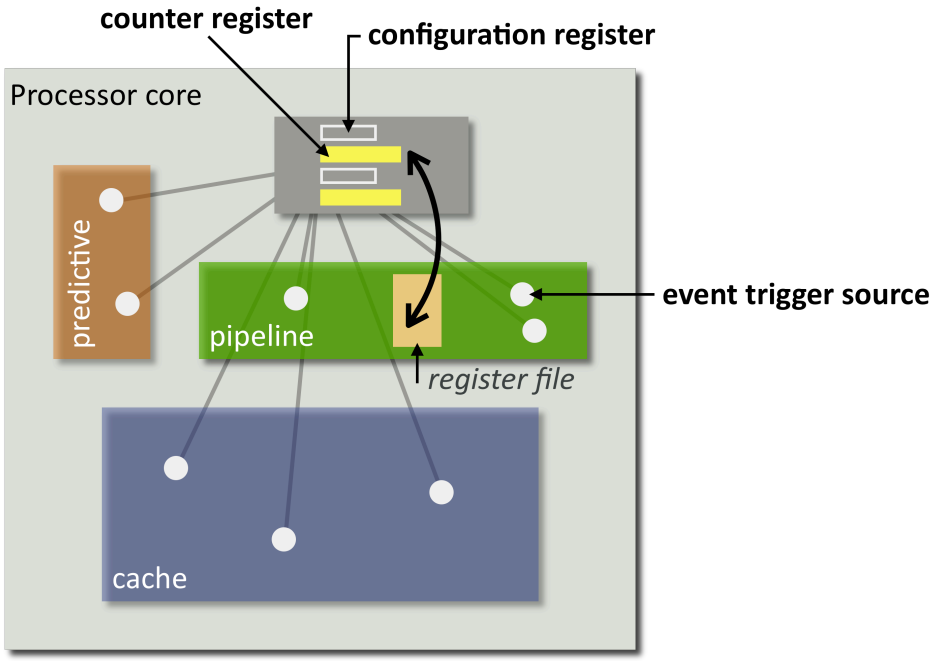


Figure 6.3: Performance counter microarchitecture

Event trigger sources exist throughout a processor, sometimes accompanied by masking configuration registers. The counters and their configuration registers are usually located together, and connected to the register file.

Event trigger sources: Event trigger sources exists throughout most processor components, and entail very little logic. In many cases, all that is required is a single-bit wire connected to the performance counter logic to indicate on which cycles a given event has occurred. Some sources may report more than one event per cycle, and require a multi-bit connection to the performance counters (e.g., instruction retirements). Finally, some event trigger sources report a type identifier that may be used by the counter to mask some subset of events. For example, a cache eviction event trigger source may want to differentiate evictions due to coherency to allow the performance counter to count just this subset of events.

Operating system support: The operating system must cooperate with the performance counter hardware in two ways. The first relates to a programmer's decision either to perform system-wide measurement, or to perform measurement of a single process.

System-wide measurement simply requires the privilege level mask to be set to allow all privilege levels to be measured.

The operating system must save and restore all configuration and counter state upon each context switch to allow measurement of a single process. The operating system support is necessary because there is no concept of a process ID that the performance counters can use as a mask in an operating system independent way.

The second reason that the operating system must cooperate with the performance counter hardware is to implement the interrupt service routine that is executed when a performance counter overflows and is configured to trigger an interrupt. The service routine will perform different functions depending on whether profiling or direct measurement will be performed.

Instruction set: Special purpose instructions are used to control and query performance counters. These instructions are privileged since the performance counters are not virtualized for each process. As a result, performance counter software libraries typically require kernel level modifications.

Instructions are required to read and write values to the configuration registers. Additionally, instructions are required to read and write values to the counter registers. Often, these instructions are implemented as register-to-register move instructions. For read operations, the source would be the machine specific configuration or counter register, and the destination would be the general-purpose register file. Writes would be implemented by reversing the source and destination.

Usage modes: Programmers can use performance counters for *time and event profiling*, *direct measurements* of execution intervals, and *performance tracing*. These usage models

provide different performance data to the programmer interested in optimizing an application.

The simplest usage of performance counters is to use them to merely count the quantity of execution events during an application's execution or an interval of time. This informs the user of the presence and magnitude of various execution events. However, the precise execution location of many execution events cannot be directly determined at a resolution better than millions of cycles due to the disruptive effects of polling the performance counters [Korn, Teller, and Castillo 2001; Maxwel et al. 2002; Callister 2006]. The execution of the measurement code introduces residual effects in the processor such as empty queues and altered caches. In addition, external interrupts and context switches can introduce more errors into measurements. We quantify this resolution limit of performance counters due to these measurement errors in Section 6.3.

Profiling: Precise counter based sampling has overcome the above shortcoming to attribute events accurately to *static program-counter locations* in a program [Dean et al. 1997; Sprunt 2002, Fields et al. 2004; Sander et al. 2006]. Sampling relies upon periodic interrupts of execution, where the interruption is triggered by the overflow of a selected program counter (PC). A performance counter is programmed to trigger an interrupt when its value first exceeds a set threshold. Each time an overflow interrupts occurs, the program-counter location is logged to determine the static program location. After numerous interrupts, a distribution of the most common, or time consuming, portions of a program are determined. When the cycle performance counter is used as the overflow interrupt trigger, the sampling is called time-based sampling, and the resulting data indicates where execution is concentrated. When other performance counters are used, such as cache misses, the sampling is

called event-based sampling, and the resulting data indicates which program locations incur many of the specified event.

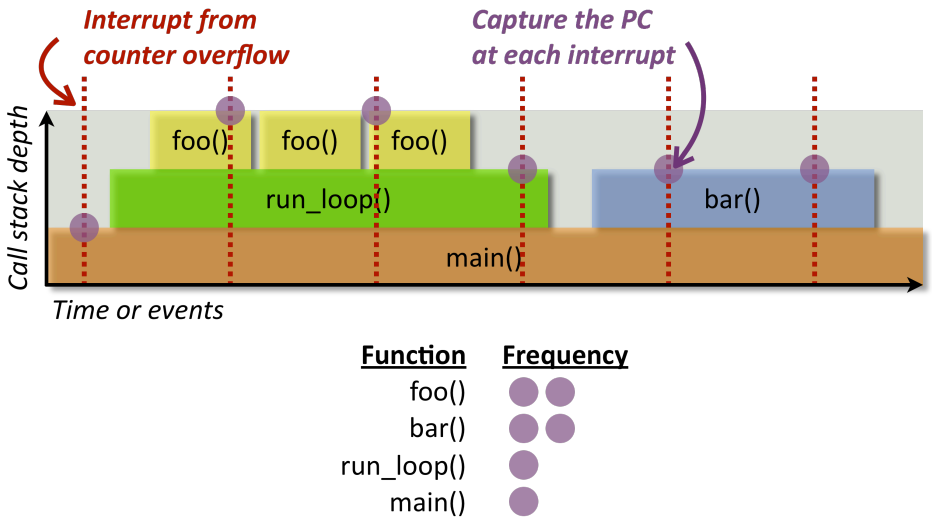
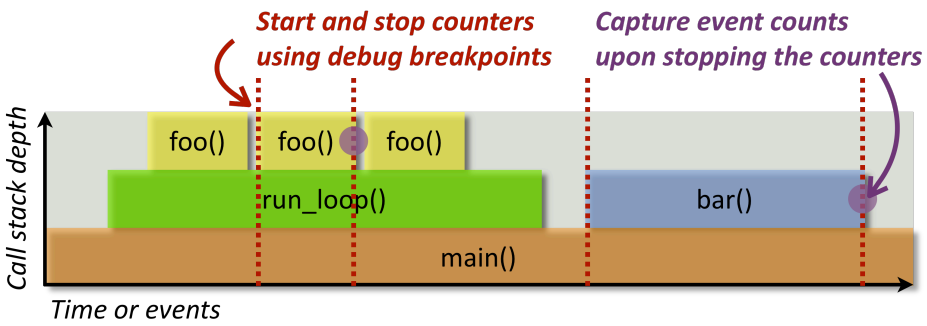


Figure 6.4: Profiling using performance counter overflow interrupts

The resulting data set provides the programmer with an approximate breakdown of execution time (or events). Programmers can target optimization efforts on functions that make up a significant proportion of execution time or a significant proportion of execution events.

Direct measurement: Direct measurements of program execution intervals are simple in comparison to profiling. Performance counters are reset at a chosen starting point within a program, and their values read at a chosen ending point. The entire execution of a program, or just a particular interval, can be measured. The programmer can simultaneously capture data from as many performance counters that exist for a given processor. Direct measurement data can indicate that certain performance degrading events are occurring at high frequency, thus directing the type of optimization efforts. Figure 6.5 illustrates direct measurements taken with performance counters.



Function	Event	Count
foo()	cycles	1,113
foo()	L2 misses	42
bar()	cycles	3,112
bar()	L2 misses	19

Figure 6.5: Direct measurements of intervals using performance counters

The resulting data set provides the programmer with metrics that can suggest which properties of the program to optimize (e.g., data locality, code size, register allocation, code scheduling).

External tools can measure an entire program's execution. However, a programmer calls start and stop performance counter library functions within a program to chose a smaller interval to measure. An alternative to modifying a program's source code is to use the debug breakpoint capabilities of modern processors. Breakpoints can be configured at starting/stopping PCs or specific data address accesses within a program. The breakpoint interrupt then starts/stops the performance counters.

Tracing: Collecting consecutive direct measurements produces a performance trace. Measurement boundaries are created by execution interrupts on performance counter overflows. Program execution is interrupted just as in profiling, but performance counter values are captured during each interrupt (like in direct measurements). We attribute events to *dynamic locations* in a program's execution. Our methods improve resolution and error tolerance beyond current limits using redundant, overlapping performance counter measurements of lower resolution with non-coincidental endpoints.

The main distinction between static and dynamic program locations is the usage of the PC versus an instruction counter respectively to determine program location. The PC defines which line of code a program's execution is at (it is also possible to determine the call stack during a measurement interrupt). We use an instruction performance counter to report the dynamic instruction count since the start of measurement. The instruction count provides a dynamic program location, for example we can distinguish the first and last iterations of a loop, something that cannot be done by examining the PC and call stack.

6.1.2 Sources of error

We now consider the sources of error that will affect the measurements collected for super-resolution reconstruction..

Ground truth: Our target of zero error (the ground truth) is when, for repeated executions of a program, the measured performance traces are identical for all possible measurement boundaries.

Total error: The total error in our measurements will be the sum of the performance counter implementation error sources described in this section plus all nondeterministic effects in the computer system (effects that differ between repeated executions). Performance counter implementation error imparts error relative to the number of measurements taken. Longer interrupt intervals can amortize these sources of error. All nondeterministic effects impart error at a constant average rate. Longer interrupt intervals will not amortize nondeterministic error.

Hardware error sources: There are two types of error due to the hardware design of processors. First, using performance counters results in some overhead and inaccuracies

due to their implementation. Second, processors are not perfectly deterministic, leading to some error relative to our ground truth target of exactly repeatable performance.

1. Interrupt service routine overhead: The first source of error for profiling is the performance impact of the interrupt service routine. An interrupt requires hundreds to thousands of instructions to execute in a modern operating system. As a result, the cache, branch predictor, and other predictive structures are polluted upon the exit of the interrupt service routine. The pollution of these performance-enhancing structures alters the performance of the program being measured [Dongarra et al., 2003]. The altered performance is error when we are interested in profiling the performance of an application. The only solution is to ensure the interrupt interval is large enough to amortize the effects of the interrupt service routine.
2. Overflow event to instruction capture delay: The second source of error for profiling is that the PC captured upon a performance counter overflow is usually not the PC of the instruction that incurred final overflow event. The PC may be for a preceding instruction when the events being measured occur early in the pipeline. On the other hand, the overflow interrupt may not be handled immediately, resulting in a PC that follows the instruction that caused the overflow. This unknown latency prevents accurate identification of instructions that cause important events such as L2 cache misses.
3. Nondeterministic hardware performance: Executing the same instructions on a processor, with the same starting microarchitectural state, does not always result in the same performance. The nondeterministic behavior of a processor is due to asynchronous events that affect performance. For example, memory bus traffic from other devices and DRAM timing make main memory access latencies nondeterministic. Another example is thermal throttling control logic that slows a processor's

clock speed when overheating occurs. Nondeterministic performance of the processor hardware results in small, but non-zero amounts of constant average rate error. Running nothing else, or a very steady state load, simultaneously with the execution thread of interest, can minimize these effects.

4. Shared execution resources: Multi-core processors that have shared resources like L2 caches, or simultaneous multithreading pipelines will have context sensitive performance that depends on the execution of other threads side-by-side with the execution thread of interest. Running nothing else, or a very steady state load, simultaneously with the execution thread of interest, can minimize these effects.

Operating system error sources: Operating system architecture directly affects the performance cost of interrupt service routines and context switches. Low cost interrupt service routines reduce the error impact of each interrupt on the measured program. Low cost context switches, and good processor affinity for executing processes minimizes the constant average rate error from other processes.

Application error sources: Applications that have a deterministic instruction execution path for a given input are well suited to super-resolution reconstruction. Applications that have nondeterministic execution paths will not have consistent performance traces between repeated executions. Examples of nondeterministic applications include: multi-threaded programs with synchronization, self-tuning programs, networked programs, and any program that depends on input from outside asynchronous systems. Effective super-resolution reconstruction for nondeterministic, and multi-threaded applications is future work.

6.1.3 Image super-resolution

Electronic imaging applications often demand high-resolution data, in other words, high pixel densities. Super-resolution image reconstruction, a.k.a. resolution enhancement, is a widely-researched technique that improves the resolution of images beyond the limits of the capturing device by combining data from multiple images of the same subject. The set of images must differ in some way to contain the additional information used in the super-resolution reconstruction. For example, the images must be offset, have varying perspectives, differing focus, etc. to enable super-resolution results that are better than the source images. Super-resolution methods have been developed for 1D signals, 2D images, and video. However, 2D image super-resolution is the primary focus of the signal processing research field [Park et al. 2003].

There are two phases of interest in super-resolution reconstruction: first, registering (a.k.a. aligning) the low-resolution input images, and second, fusing the inputs to produce a high-resolution output image. Registration involves computing the relative positions of each input image so that they can be overlaid with precision. The relative offsets may be known or unknown depending on the source of the low-resolution images. There are techniques for estimating the offsets if they are unknown. If the offsets are known, then registration is simply a matter of shifting the source images. (In hardware performance counter super-resolution, precise registration is essentially known, as we will describe later.)

The second step in image super-resolution is fusion. This step usually comprises of both deblurring and reconstruction steps. Deblurring is often used because of the assumption of limited aperture when capturing low-resolution data, but we are not concerned about this problem in the context hardware performance counters. Reconstruction takes multiple low-resolution "pixels," with varying sub-pixel offsets, and solves for the underlying high-

resolution, i.e., smaller, pixels. In the context of our work, the low-resolution raw counter measurements correspond to the low-resolution pixels, and the high-resolution performance profile corresponds to the reconstructed high-resolution image.

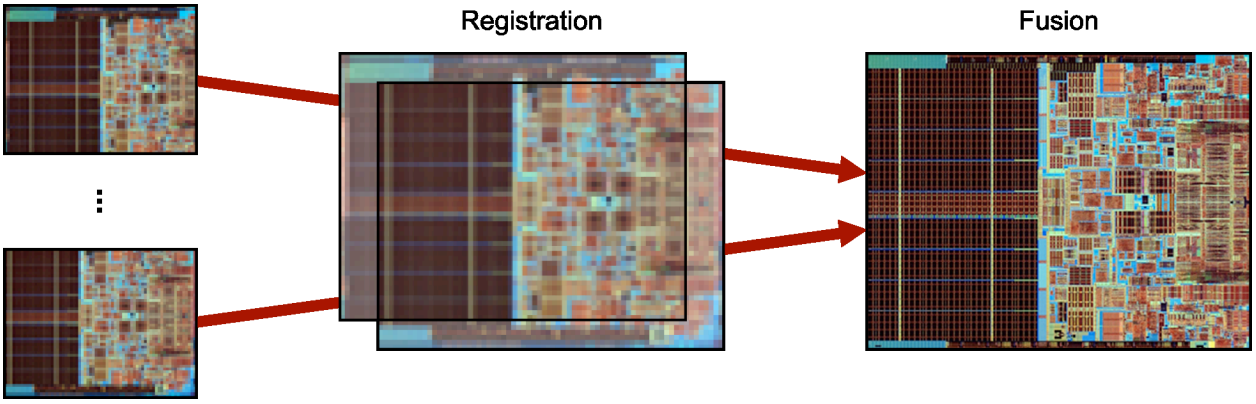


Figure 6.6: Image super-resolution reconstruction

Image super-resolution takes several images captured at low resolution, registers (a.k.a. aligns) them, and then fuses them into a high-resolution image. (Note: Resolution differences and number of captured images have been exaggerated for illustration purposes.)

There are many proposed algorithms for the fusion phase of super-resolution. The algorithms can be roughly categorized as interpolation approaches, frequency domain approaches, iterative approaches, stochastic approaches, and filtering approaches. Each class of algorithm makes different assumptions about the expected high-resolution result to deal with the ill-posed problem of too few inputs compared to the output. Our approach for performance counters is based on the iterative approaches because they can use the most flexible constraints.

General-purpose super-resolution algorithms produce mixed results, and in practice, can achieve maximum resolution increases of less than ten-times magnification. In addition, the algorithms tolerate only small amounts of random noise [Baker and Kanade 2002].

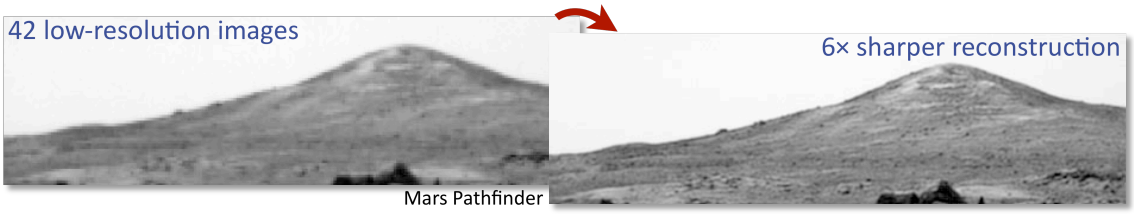


Figure 6.7: Image super-resolution results

[NASA]

At least in theory, applying existing image processing super-resolution algorithms to performance counter measurements only requires modifying the algorithms to work in a single dimension instead of two dimensions. However, existing super-resolution algorithms for image processor do not meet our super-resolution requirements, so we developed our own method that is suited to the properties of performance counter measurements. Our algorithm is based on a formulation in linear programming as described later.

6.1.4 Performance trace notation and variables

Matrix notation: Matrices are rectangular arrays of real numbers. We represent matrices with bold, upper case, letters. Matrix elements are identified by their row and column position, *i* and *j*, respectively.

$$\mathbf{A}_{i,j}$$

The size (order) of a matrix is defined as *m* rows by *n* columns.

$$[m \times n]$$

Vector notation: Vectors are ordered columns of real numbers. Vectors are represented by bold, lower case, letters. Row vectors are represented by the transpose of a column vector. Vector elements are identified by their position *i*.

$$\mathbf{b}_i$$

The size (dimension) of a vector is defined as m rows.

$$[m]$$

We indicate vectors that belong to a lexicographic set with a superscript index k.

$$\mathbf{b}^{(k)}$$

We define several vector to scalar operations as follows.

$\|\|\mathbf{b}\|\|_1$	$\mathrm{L}^1\ \mathrm{norm}(\mathbf{b}) = \sum_i \|\mathbf{b}_i\|$	manhattan distance
$\|\|\mathbf{b}\|\|_2$	$\mathrm{L}^2\ \mathrm{norm}(\mathbf{b}) = \sqrt{\sum_i \mathbf{b}_i^2}$	euclidean distance
$\bar{\mathbf{b}}$	$\mathrm{mean}(\mathbf{b}) = \frac{1}{m}\sum_i \mathbf{b}_i$	arithmatic mean
$\sigma_{\mathbf{b}}^2$	$\mathrm{var}(\mathbf{b}) = \sum_i \left(\mathbf{b}_i - \bar{\mathbf{b}}\right)^2$	variance
$\sigma_{\mathbf{b}}$	$\mathrm{stdev}(\mathbf{b}) = \sqrt{\sigma_{\mathbf{b}}^2}$	standard deviation
$\mathrm{V}_{\mathbf{b}}$	$\mathrm{CV}(\mathbf{b}) = \sigma_{\mathbf{b}} / \bar{\mathbf{b}}$	coefficient of variance
$\mathrm{M}_{\mathbf{b}}$	$\mathrm{median}(\mathbf{b}) = Q_2(\mathbf{b})$	2nd quartile

We define abbreviated notations for vector conditional tests.

$\mathbf{b} == \mathbf{c}$	all elements equality	$\mathbf{b}_i = \mathbf{c}_i\ \forall\ i$
$\mathbf{b} \geq \mathbf{0}$	all elements non-negative	$\mathbf{b}_i \geq 0\ \forall\ i$

We define a lexicographic vector set to vector operation as follows.

$$\mathbb{M}_{\mathbf{b}^{(k)}} \quad \mathrm{medianVector}\left(\mathbf{b}^{(k)}\right) = \mathrm{M}\left(\mathbf{b}_i^{(1)}, \ldots, \mathbf{b}_i^{(K)}\right) \forall\ i$$

Performance counter variables: We represent performance trace data as a vector of *measurements* with several associated metadata vectors.

Table 6.1: Performance counter vectors

variable	name	units of elements
$\mathbf{m}$	measurements	events
$e\mathbf{m}$	measurement error	events
$s\mathbf{m}$	start position of measurements	instructions
$\ell\mathbf{m}$	length of measurements	instructions
$b\mathbf{m}$	boundaries of measurements	instructions

Captured measurements are nonnegative, but can have positive or negative error relative to the *ground truth* (GT). Error is introduced by the execution interruptions necessary to capture measurements as well as the non-deterministic performance of repeated execution of the same instruction stream.

$$\mathbf{m} = \mathbf{m}^{\mathrm{GT}} + e\mathbf{m} \quad where \quad \mathbf{m} \geq 0\,,\ \mathbf{m}^{\mathrm{GT}} \geq 0$$

An effective measure of the quality of a measurement vector is its signal-to-noise ratio, often expressed in decibels.

$$\mathrm{SNR}_{\mathbf{m}} = 10 \log_{10}\left(\frac{\sigma^2_{\mathbf{m}^{\mathrm{GT}}}}{\sigma^2_{e\mathbf{m}}}\right) = 20 \log_{10}\left(\frac{\sigma_{\mathbf{m}^{\mathrm{GT}}}}{\sigma_{e\mathbf{m}}}\right)$$

The SNR indicates the relative magnitude of the GT "signal" to the error. SNR is effective at measuring usefulness to a user of measurement data because SNR is correlated to the accuracy of outlier identification. A SNR of 0 dB indicates equal magnitude standard deviation of the GT and error, 20 dB indicates 10 times greater GT standard deviation, 40 dB indicates 100 times greater GT standard deviation, and -20 dB indicates 10 times smaller GT standard deviation.

6.2 Implementation

An *experiment* consists of a performance trace of repeated executions of the same instruction stream and all the corresponding properties. The measured instruction stream is the complete instruction stream of an application under analysis or a limited window of the application's instruction stream.

When we measure Y iterations of a complete application,

$$\sum_{i=1}^{N} \ell \mathbf{m}_i = Y \times \ell^{\text{iteration}}$$

where $\ell^{\text{iteration}}$ is the length of each iteration of the application's execution in retired instructions and N is the number of captured measurements.

We can define an estimate of the ground truth $\mathbf{m}^{\text{gt}}$ if we capture K repeated experiments with the same boundaries.

$$\mathbf{m}^{(k)} = \left\{\mathbf{m}^{(1)},\ \ldots,\ \mathbf{m}^{(K)}\right\} \quad where \quad b\mathbf{m}^{(k)} == b\mathbf{m}^{(k)}\ \forall\ k$$

$$\mathbf{m}^{\text{gt}} = \mathbb{M}\mathbf{m}^{(k)}$$

We formally define the ground truth as the median vector of "all possible" experiments, i.e., infinite repeated experiments.

$$\mathbf{m}^{\text{GT}} = \mathbb{M}\mathbf{m}^{(k)} \quad where \quad K = \infty$$

We define SNR more precisely in the presence of K experiments.

$$\text{SNR}_{\mathbf{m}^{(k)}} = 10 \log_{10}\left(\frac{\sigma^2_{\mathbf{m}^{\text{GT}}}}{\sigma^2_{e\mathbf{m}^{(k)}}}\right)$$

Regions: We wish to reconstruct high-resolution measurements from low-resolution captured measurements. We call the high-resolution measurements *regions* to distinguish them from the low-resolution measurements. Thus, super-resolution reconstruction solves the number of events in high-resolution regions given low-resolution measurements.

We represent the solved region events as vector **r**, along with the corresponding meta-data vectors $e\mathbf{r}$, $\ell\mathbf{r}$, and $b\mathbf{r}$. Our reconstruction objective,

$$\text{minimize } \|e\mathbf{r}\|_1 \quad \textit{where} \quad \mathbf{r} \geq 0\,,\ \mathbf{r}^{\text{gt}} \geq 0\,,\ \mathbf{r} = \mathbf{r}^{\text{gt}} + e\mathbf{r}$$

The primary relationship between regions and measurements if represented by μ for magnification.

$$\mu = \frac{\overline{\ell\mathbf{m}}}{\overline{\ell\mathbf{r}}} \quad \textit{typically} \quad \mu > 1$$

When we measure a complete application, we can solve for V regions such that,

$$\sum_{j=1}^{V} \ell\mathbf{r}_j = \ell^{\text{iteration}}$$

Finally, the user's objective may be to minimize mean region length while meeting a minimum $\text{SNR}_{\mathbf{r}}$.

$$\text{SNR}_{\mathbf{r}} = 10 \log_{10}\left(\frac{\sigma^2_{\mathbf{r}^{\text{GT}}}}{\sigma^2_{e\mathbf{r}}}\right)$$

There are other possible user objectives, but all user objectives require selecting an effective combination of μ and Y.

Characterization variables: We introduce ℓ to represent the measurement length, also referred to as the "resolution" of an experiment. For a given experiment,

$$\ell = \overline{\ell \mathbf{m}}$$

We label the ground truth event rate as $\mathcal{R}^{\mathrm{GT}}$. The mean number of events per measurement as a function of measurement length should be directly proportional to the mean event rate if there is no measurement error.

$$\overline{\mathbf{m}}(\ell) = \mathcal{R}^{\mathrm{GT}} \times \ell \quad when \quad e\mathbf{m} = 0$$

$$\mathcal{R}^{\mathrm{GT}} = \frac{\overline{\mathbf{m}^{\mathrm{GT}}}}{\ell}$$

Measurement error has two components, interrupt error and steady state error as we describe in Section 6.2.2.

$$e\mathbf{m} = e^{\text{interrupt}}\,\mathbf{m} + e^{\text{steady state}}\,\mathbf{m}$$

The mean of steady state error is near zero because this error is relative to the ground truth (the median of nondeterministic performance of repeated executions). The mean of interrupt error is positive because measurement interrupts typically cause extra events to occur during application execution.

$$\overline{e^{\text{steady state}}\,\mathbf{m}} \simeq 0\,,\ \overline{e^{\text{interrupt}}\,\mathbf{m}} \geq 0$$

We model mean measurement error as a linear function that we empirically verifiy in Section 6.3.1.

$$\overline{\mathbf{m}}(\ell) = \mathcal{R}^{\mathrm{GT}} \times \ell + \overline{e^{\text{interrupt}}\,\mathbf{m}}$$

First, we measure $\mathcal{R}^{GT}$ and the mean measurement events for multiple values of ℓ. Then we solve for the mean measurement interrupt error using regression. Deviation from the regression indicates destructive interference from measurement interrupts. The resolution where this deviation becomes severe is the resolution limit (for median reconstruction) in conjunction with SNR.

As we reconstruct super-resolution results we wish to estimate the magnitude of error of the solved regions. When calculating the median/mean inverse of ill-conditioned linear systems with constant point-spread functions, we can establish the approximate relationship between measurement error and reconstruction error:

$$\sigma_R^2 = \frac{\mu^2}{N}\sigma_M^2 + \overline{\left\|\nabla M_{gt}\right\|^2}\sigma_{reg}^2$$

where μ is magnification, and N is the number of iterations. The first term is the event error, the second term is the registration noise. The second term is the registration error that is the mean gradient energy of the ground truth times the registration error variance [Pham, van Vliet, and Schutte 2005]. Thus, we can roughly estimate the reconstructed SNR:

$$SNR_R = 20\log\left(\frac{\mathrm{var}\left(R_{gt}\right)}{\sigma_R^2}\right) = 20\log\left(\frac{\mathrm{var}\left(R_{gt}\right)}{\mathrm{var}\left(R_1 - R_{gt}\right)}\right)$$

The second form indicates that we can determine SNR_R experimentally, without a known ground truth, using k-fold cross-validation.

It is interesting to note that the median is the central point that minimizes absolute deviations, i.e., the $L1$ norm, also, there exists a GLRT test that distinguishes if $L1$ or $L2$ norm is superior based on better fit of Laplacian or Gaussian PDF [Farsiu 2003].

Our reconstruction problems are underconstrained, which does not prevent an LP solution, but does not provide stable, or even desirable results. We selected a smoothing function compatible with linear programming: bilateral total variation (BTV) regularization [Farsiu 2004]:

$$\mathrm{BTV}(R) = \sum_{x=-P}^{P} \alpha^{|x|} \left\| R - S^x R \right\|_1$$

where S^x is the shift operator by x units. Parameter P determines the number of resolution scales to compute derivatives, usually set P=2. Parameter α determines the fall-off effect, 0<α<1, and usually set α=½. This reduces the BTV function to:

$$\mathrm{BTV}(R) = .25\left\| R - S^{-2}R \right\|_1 + .5\left\| R - S^{-1}R \right\|_1 + .5\left\| R - S^{1}R \right\|_1 + .25\left\| R - S^{2}R \right\|_1$$

which we convert to the second objective function of a lexicographical goal programming model.

6.2.1 Platform

Super-resolution reconstruction is most applicable to computational kernels that a programmer would want to optimize for performance on a particular processor. Computational kernels are the performance critical portions of scientific computing, multimedia data processing, and communications applications. They are often written in C or assembly code for maximum performance, despite the increased programming effort, due to their pervasive reuse.

We focused our study on obtaining performance traces of an array of numerical application kernels from the Intel Performance Primitives (IPP) library. The performance of these

kernels is typically compute bound, and their performance has high value to many applications. Thus, the kernels are hand tuned for individual computer platforms on which they will be executed. Performance traces are useful for further performance exploration and tuning of both the kernels and the CPU microarchitectures that execute them.

The IPP library provides hundreds of numerical functions in a broad array of application domains. We selected a representative subset of functions for study from fourteen distinct domains, that process integer and floating-point vectors, matrices, 1D signals, 2D images, and 3D rendering geometry: vector math, small matrix, audio coding, data compression, signal processing, speech coding, speech recognition, strings, color conversion, computer vision, image processing, JPEG, video coding, and rendering.

We performed most of our measurements using 20 of the most important performance counters of the Intel Core 2 CPU architecture. The performance counters can measure over 200 architectural and microarchitectural performance events, including pipeline events, memory events, and predictive structure events. Experiments were also performed using dynamically instrumented binaries, and synthetically generated data.

Table 6.2: Selected Intel Core 2 performance counters

Category	Definition (pfmon name)	Description
Architectural	Instructions retired (instructions_retired)	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
Architectural	Instructions retired, which contain a load (inst_retired:loads)	This event counts the number of instructions retired that contain a load operation.
Architectural	Instructions retired, which contain a store (inst_retired:stores)	This event counts the number of instructions retired that contain a store operation.
Architectural	Retired branch instructions	This event counts the number of branch instruc-

Category	Definition (pfmon name)	Description
	(branch_instructions_retired)	tions retired.
Architectural	Retired streaming SIMD instructions (simd_inst_retired:any)	This event counts the overall number of SIMD instructions retired.
Time	Core cycles when core is not halted (unhalted_core_cycles)	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason, this event may have a changing ratio in regard to time. When the core frequency is constant, this event can give approximate elapsed time while the core not in halt state.
Pipeline	Cycles during which the instruction queue is full (inst_queue:full)	This event counts the number of cycles during which the instruction queue is full. In this situation, the core front-end stops fetching more instructions. This is an indication of very long stalls in the back-end pipeline stages.
Pipeline	Cycles during which the reservation station is full (resource_stalls:rs_full)	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation, new instructions can not enter the pipe and start execution.
Pipeline	Cycles during which the pipeline has exceeded the load or store limit or is waiting to commit all stores (resource_stalls:ld_st)	This event counts the number of cycles while resource-related stalls occur due to: • The number of load instructions in the pipeline reached the limit the processor can handle. The stall ends when a loading instruction retires. • The number of store instructions in the pipeline reached the limit the processor can handle. The stall ends when a storing instruction commits its data to the cache or memory. • There is an instruction in the pipe that can be executed only when all previous stores complete and their data is committed in the caches or memory. For example, the SFENCE and MFENCE instructions require this behavior.

Category	Definition (pfmon name)	Description
Pipeline	Cycles while stores are blocked due to store buffer drain (sb_drain_cycles)	This event counts every cycle during which the store buffer is draining. This includes: • Serializing operations such as CPUID • Synchronizing operations such as XCHG • Interrupt acknowledgment • Other conditions, such as cache flushing
Memory	Instruction TLB misses (itlb:misses)	This event counts the number of instruction fetches from either small or large pages that miss the ITLB.
Memory	Instruction fetch unit misses (l1i_misses)	This event counts all instruction fetches that miss the instruction fetch unit or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once, not once for every cycle it is outstanding.
Memory	Loads blocked by the L1 data cache (load_block:l1d)	This event indicates that loads are blocked due to one or more reasons. Some triggers for this event are: • The number of L1 data cache misses exceeds the maximum number of outstanding misses supported by the processor. This includes misses generated as result of demand fetches, software prefetches or hardware prefetches. • Cache line split loads. • Partial reads, such as reads to uncacheable memory, I/O instructions and more. • A locked load operation is in progress. The number of events is greater or equal to the number of load operations that were blocked.
Memory	Cycles while store is waiting for a preceding store to be globally observed (store_block:order)	This event counts the total duration, in number of cycles, which stores are waiting for a preceding stored cache line to be observed by other cores. This situation happens as a result of the strong store ordering behavior. The stall may occur and be noticeable if there are many cases when a store either misses the L1 data cache or hits a cache line in the Shared state. If the store requires a bus transaction to read the cache line then the stall ends when snoop response for the bus transaction arrives.
Memory	Memory accesses that missed the data TLB (dtlb_misses:any)	This event counts the number of DTLB misses. The count includes misses detected as a result of speculative accesses. Typically a high count for this event indicates that the code accesses a large number of data pages.
Memory	L2 cache demand requests from this core	This event counts all completed L2 cache demand requests from this core. This includes

Category	Definition (pfmon name)	Description
	(last_level_cache_references)	L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches.
Memory	L2 cache demand requests from this core that missed the L2 (last_level_cache_misses)	This event counts all completed L2 cache demand requests from this core that miss the L2 cache. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches.
Memory	All bus transactions for this core (bus_trans_any:self)	This event counts all bus transactions for this core. This includes: • Memory transactions • IO transactions (non memory-mapped) • Deferred transaction completion • Other less frequent transactions, such as interrupts
Predictive	Retired mispredicted branch instructions (mispredicted_branch_retired)	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa.
Predictive	L1 data cache prefetch requests (l1d_prefetch:requests)	This event counts the number of times the L1 data cache requested to prefetch a data cache line. Requests can be rejected when the L2 cache is busy and resubmitted later or lost. All requests are counted, including those that are rejected.

We collect performance trace measurements using the perfmon2 project on Linux. We use Python scripts to setup and initiate the performance trace that is performed by the pfmon tool of perfmon2. The same scripts also capture and convert the performance trace output to a simple tab-separated format suitable for analysis in statistical software (e.g., R) and for performance counter super-resolution.

We use a common file format to represent measured performance counter traces, median reconstructed traces, and super-resolution reconstructed traces. Each file that follows our format is called an 'event trace' file. The trace format is based upon ASCII-encoded text with tab and newline delimiters.

A trace file has four sequential sections called type, metadata, field attributes, and field data. Each section has a unique line format that allows identification of the start of each section. We document the trace file syntax using Extended Backus-Naur Form.

Table 6.3: Performance trace file format BNF

Symbol		Expression
event-trace-file	=	type-section metadata-section field-attribute-section field-data-section
type-section	=	type NEWLINE
type	=	“measurements” | “measurements (synthetic) | “measurements (median)” | “regions”
metadata-section	=	iteration-length {metadata}
iteration-length	=	“iteration length” TAB INTEGER NEWLINE
metadata	=	name TAB value NEWLINE
field-attribute-section	=	field-names {field-attribute}
field-names	=	“events” TAB “start” TAB “length” {TAB name} NEWLINE
field-attribute	=	name TAB TAB {TAB value} NEWLINE
field-data-section	=	{measurement}
measurement	=	TAB INTEGER TAB INTEGER {TAB number} NEWLINE
name	=	STRING
number	=	INTEGER | FLOATING-POINT
value	=	STRING | INTEGER | FLOATING-POINT

(* *NOTE: all* field-names, field-attribute, *and* measurement *expressions must have an equal number of* name, value, *and* number *fields, respectivly* *)

Visualization: We have created a graphical user interface to display the collected measurements, allow a user to specify fusion parameters, and view the resulting super-resolution reconstruction. The rapid feedback of results as we change the fusion parameters has been a great help in designing our linear programming formulation and debugging the entire super-resolution approach. The tool is written in Java, and uses the Mosek library (written in C) for the linear programming optimizations.

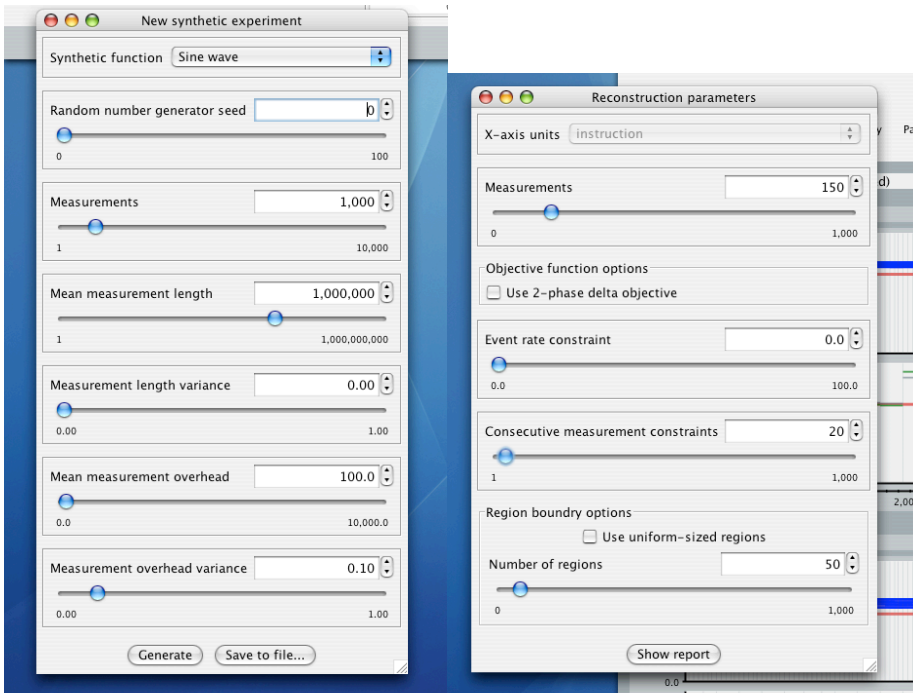


Figure 6.8: Synthetic trace and reconstruction parameter controls

Our visualization tool allows the generation of synthetic data sets for study (sine waves, impulse functions, and instruction mixes from 2^{10}, 2^{15}, and 2^{19}-point DFTs). In addition, there are several parameters for the fusion/reconstruction computation. The controllable parameters are shown.

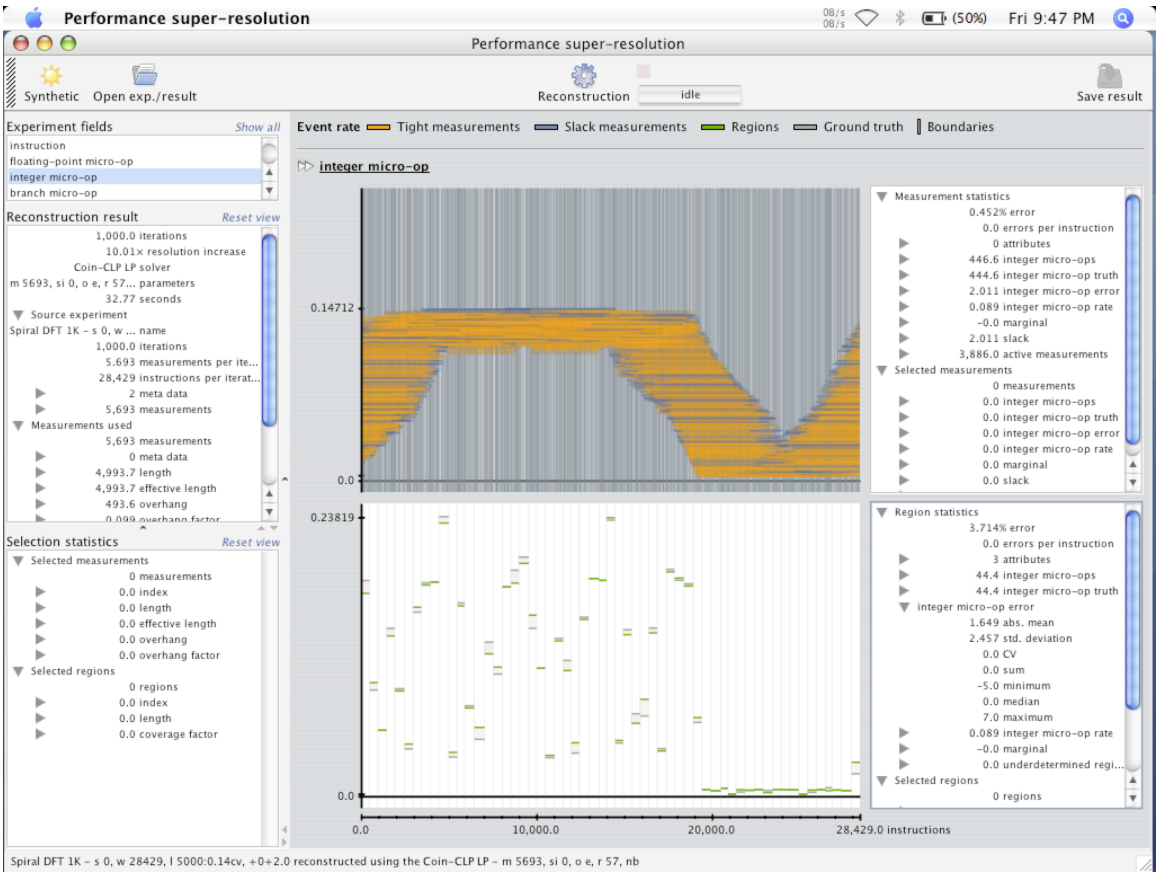


Figure 6.9: Main window of visualization tool

The top plot shows the branch instruction measurements (blue) taken for a portion of a 2^{15}-point DFT. The horizontal axis is the dynamic instruction location of each measurement in SSE2 instructions. The vertical axis is the rate of branch instructions per SSE2 instructions. Each measurement is 1 million SSE2 instructions long. The bottom plot shows the reconstruction results (green) and the reference/correct results (grey) for regions with an average length of 116,163 SSE2 instructions, for a magnification factor of 8.6 times. We used 150 measurements taken from 26 repeated executions of the DFT, and the final reconstruction error is 2.5%. (Note: See Figure 6.8 for the reconstruction parameters used).

6.2.2 Error characterization

The error imparted by all sources falls into two categories: measurement interrupt error and steady state error (constant average rate error). In addition, both categories are variable throughout a program's execution. Thus, both the mean value and variance of measurement interrupt error and steady state error are of interest. Table 6.4 summarizes these four error properties, and their sources.

Table 6.4: Performance counter error categories and sources

Measurement interrupt error is amortizable, but longer measurements do not affect steady state error.

Error type	Component	Sources
Measurement interrupt error	fixed amount	interrupt service routine overhead overflow event to instruction capture delay
Measurement interrupt error	variable	interrupt service routine overhead overflow event to instruction capture delay
Steady state error	fixed rate	preemptive multitasking context switches*
Steady state error	variable	nondeterministic hardware performance shared execution resources external interrupts* voluntary context switches* nondeterministic execution paths

* sources of large error outliers

Sensitivity to measurement length: We expect that increasing the interval between overflow interrupts will amortize measurement interrupt error. Thus, we expect total error to decrease as we increase measurement length. We cannot amortize steady state error, therefore making measurements longer will never decrease total error below the steady state error rate.

Individual measurement error distribution: We expect that most measurements will have low error, with low variance. However, there will be large outliers (many standard deviations from the mean) due to infrequent, but large impact, occurrences that disrupt performance. Specifically, external interrupts and non-measurement related context switches are the major sources of this type of behavior due to their pollution of microarchitectural state.

Sensitivity to platform: We expect the general characteristics of error (sensitivity to measurement length and error distribution) to be similar across different measured programs, event trigger sources, and the hardware platforms (processor microarchitectures). The fundamental sources of error remain the same across all of these dimensions. However,

the mean, variance, and exact distribution of error are expected to vary with respect to each dimension.

6.2.3 Median reconstruction

Median reconstruction of performance traces uses multiple traces, with equal measurement offsets, to reconstruct a performance trace with a higher signal to noise ratio that the individual input traces. The measurement-wise median estimator is effective at producing stable results when given data with large and skewed outliers. A successful median reconstruction result is defined by:

1. High signal to noise ratio
2. High confidence in median estimates
3. Low input trace collection cost

We collect and reconstruct median performance traces in an adaptive fashion to achieve a desired level of confidence in the resulting trace. Input traces are collected using direct event tracing, and the confidence intervals of the median estimates are determined as each trace is added to the input set. Collection is completed when a desired fraction of measurements meet the target confidence interval, or if a maximum input trace count is reached.

The median reconstruction algorithm has three phases:

1. Acquire a performance trace using direct event tracing.
2. Compute the measurement-wise median confidence intervals using all traces.
3. Repeat steps 1 & 2 until confidence targets are met.

These steps are sufficient to compute a median reconstruction with high confidence in the result given the following requirements are met (in addition to the direct event tracing requirements):

1. The measured program has a deterministic instruction stream.
2. Measurement interrupts can be performed at reproducible offsets.
3. Six or more traces are needed to compute the median confidence intervals.

For our experiments to determine the effectiveness of median reconstruction, we needed to estimate the signal to noise ratio. The computation of the signal to noise ratio needs accurate estimates of variance. Good variance estimates need large sample sizes, adding an additional requirement, and additional algorithm steps, to our experimental methodology.

6.2.4 Super-resolution reconstruction

Performance tracing using performance counters has resolution limits when a given measurement fidelity is required. A higher fidelity, and/or higher resolution, result can be reconstructed from multiple sets of low fidelity, low resolution data. This enhancement process is called super-resolution because it produces higher resolution results than are possible via direct capture.

We apply super-resolution to performance tracing data to reconstruct lower error and higher resolution data than is directly measurable. The reconstruction should be robust to measurement noise and registration error, and require a tractable amount of computation. A successful super-resolution reconstruction result is defined by:

1. High resolution

2. High signal to noise ratio
3. Low computational cost

We reconstruct high-resolution performance traces in four major phases.

1. Acquire and register low-resolution measurements
2. Choose high-resolution regions to reconstruct
3. Solve the reconstruction program
4. Cross-validate to determine confidence of results

We acquire contiguous measurements at a frequency (resolution) that has a low proportion of noise from the interrupts necessary for measurement. Each performance measurement is made up of: the performance event count, the measurement start position, and the measurement length.

The performance event count is the integer number of execution events that occur during a measurement interval. For example, if we are measuring cache miss events, the event count is the number of cache misses that occurred between the start and end of a measurement interval.

Event counts are collected at regular intervals using performance counter interrupt on overflow. Ideally, this event count only includes events caused by the execution of the instruction stream under study, and does not include noise from the measurement process or other sources.

The measurement start position and length align (register) each measurement with respect to the execution of the instruction stream under study. The start and length are collected using the executed instruction performance counter simultaneously with the

performance event count. The executed instruction counter must provide deterministic results for repeated executions of the instruction stream under study to allow for accurate registration.

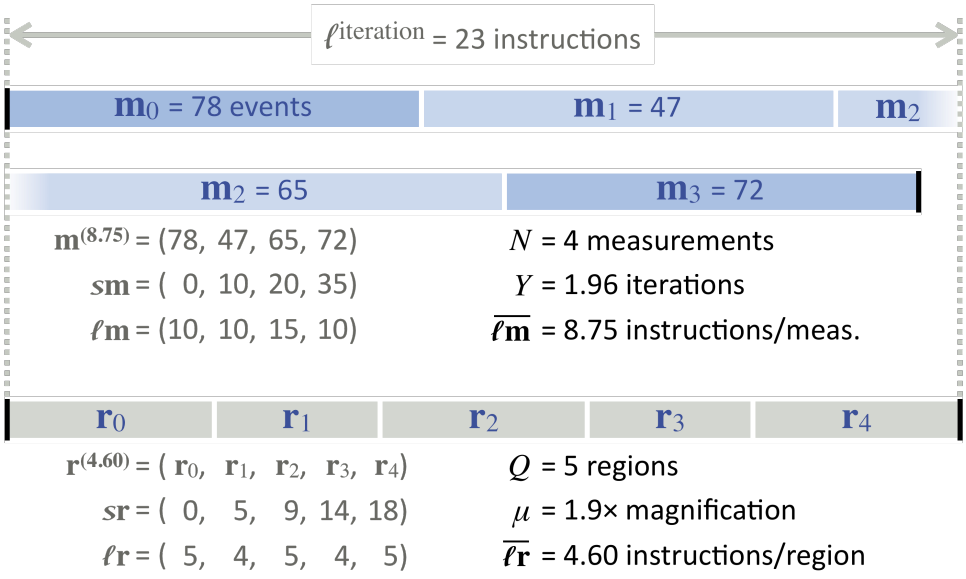


Figure 6.10: Super-resolution problem setup

An illustrative example of a super-resolution reconstruction problem, with 4 measurement inputs, solving for 5 smaller regions. This experiment has few iterations, and is therefore is under-constrained, to keep its size down for clarity.

$$\underset{\mathbf{x}}{lexmin}\ \mathbf{w}^{(k)}\ subject\ to\ \mathbf{Ax} = \mathbf{b}$$

lexicographic minimize:			*objectives*
$\mathbf{w}^{(1)}$	$= e\mathbf{m}^{+}_0 + e\mathbf{m}^{+}_1 + e\mathbf{m}^{+}_2 + e\mathbf{m}^{+}_3 + e\mathbf{m}^{-}_0 + e\mathbf{m}^{-}_1 + e\mathbf{m}^{-}_2 + e\mathbf{m}^{-}_3$		constrain structure
$\mathbf{w}^{(2)}$	$= 4.5\ d\mathbf{r}^{1+}_0 + 4.5\ d\mathbf{r}^{1+}_1 + 4.5\ d\mathbf{r}^{1+}_2 + 4.5\ d\mathbf{r}^{1+}_3 + 4.5\ d\mathbf{r}^{1-}_0 + 4.5\ d\mathbf{r}^{1-}_1 + 4.5\ d\mathbf{r}^{1-}_2 + 4.5\ d\mathbf{r}^{1-}_3 +$ $2.25\ d\mathbf{r}^{2+}_0 + 2.25\ d\mathbf{r}^{2+}_1 + 2.25\ d\mathbf{r}^{2+}_2 + 2.25\ d\mathbf{r}^{2-}_0 + 2.25\ d\mathbf{r}^{2-}_1 + 2.25\ d\mathbf{r}^{2-}_2$		regularize

subject to: $\mathbf{b}$ = (		$\mathbf{A}$ = [	*constraints*	*variables*	
"m_0"	78 =	$\mathbf{r}_0 + \mathbf{r}_1 + \mathbf{G}_{0,2} + 10\ e\mathbf{m}^{+}_0 - 10\ e\mathbf{m}^{-}_0$	measurements	i	measurement index
"m_1"	47 =	$\mathbf{G}_{1,2} + \mathbf{r}_3 + \mathbf{G}_{1,4} + 10\ e\mathbf{m}^{+}_1 - 10\ e\mathbf{m}^{-}_1$		j	region index
"m_2"	65 =	$\mathbf{G}_{2,4} + \mathbf{r}_0 + \mathbf{r}_1 + \mathbf{G}_{2,2} + 15\ e\mathbf{m}^{+}_2 - 15\ e\mathbf{m}^{-}_2$		$\mathbf{x}$ = (	
"m_3"	72 =	$\mathbf{G}_{3,2} + \mathbf{r}_3 + \mathbf{G}_{3,4} + 10\ e\mathbf{m}^{+}_3 - 10\ e\mathbf{m}^{-}_3$		$\mathbf{r}_j$	regions
				$\mathbf{G}_{i,j}$	overhang
"$r_{2,s0}$"	0 =	$\mathbf{r}_2 - \mathbf{G}_{0,2} - \mathbf{G}_{1,2}$	contiguous overhang sets	$e\mathbf{m}^{+}_i$	positive meas. error rate
"$r_{2,s1}$"	0 =	$\mathbf{r}_2 - \mathbf{G}_{2,2} - \mathbf{G}_{3,2}$		$e\mathbf{m}^{-}_i$	negative meas. error rate
"$r_{4,s0}$"	0 =	$\mathbf{r}_4 - \mathbf{G}_{1,4} - \mathbf{G}_{2,4}$		$d\mathbf{r}^{1+}_j$	pos. single region rate delta
"$r_{4,s1}$"	0 ≤	$\mathbf{r}_4 - \mathbf{G}_{3,4}$		$d\mathbf{r}^{1-}_j$	neg. single region rate delta
				$d\mathbf{r}^{2+}_j$	pos. double region rate delta
"$r_{2,s0,s1}$"	0 ≤	$\mathbf{r}_2 - \mathbf{G}_{0,2} - \mathbf{G}_{3,2}$	complimentary overhang pairs	$d\mathbf{r}^{2-}_j$	neg. double region rate delta
				)	
"d^1_0"	0 =	$\frac{1}{5}\mathbf{r}_0 - \frac{1}{4}\mathbf{r}_1 + d\mathbf{r}^{1+}_0 - d\mathbf{r}^{1-}_0$	single region rate delta		*all variables non-negative*
"d^1_1"	0 =	$\frac{1}{4}\mathbf{r}_1 - \frac{1}{5}\mathbf{r}_2 + d\mathbf{r}^{1+}_1 - d\mathbf{r}^{1-}_1$			
"d^1_2"	0 =	$\frac{1}{5}\mathbf{r}_2 - \frac{1}{4}\mathbf{r}_3 + d\mathbf{r}^{1+}_2 - d\mathbf{r}^{1-}_2$			
"d^1_3"	0 =	$\frac{1}{4}\mathbf{r}_3 - \frac{1}{5}\mathbf{r}_4 + d\mathbf{r}^{1+}_3 - d\mathbf{r}^{1-}_3$			
"d^2_0"	0 =	$\frac{1}{5}\mathbf{r}_0 - \frac{1}{5}\mathbf{r}_2 + d\mathbf{r}^{2+}_0 - d\mathbf{r}^{2-}_0$	double region rate delta		
"d^2_1"	0 =	$\frac{1}{4}\mathbf{r}_1 - \frac{1}{4}\mathbf{r}_3 + d\mathbf{r}^{2+}_1 - d\mathbf{r}^{2-}_1$			
"d^2_2"	0 =	$\frac{1}{5}\mathbf{r}_2 - \frac{1}{5}\mathbf{r}_4 + d\mathbf{r}^{2+}_2 - d\mathbf{r}^{2-}_2$			
"constraint names"	)	]			

Figure 6.11: Super-resolution reconstruction linear programming

The constraint and objective equations for the previous illustrative example.

6.3 Results

We present our initial results in performance counter error characterization and super-resolution reconstruction performance.

6.3.1 Performance counter error

We required a more detailed empirical characterization of performance counter error than previous work. In this section, we examine performance counter error when collecting

performance traces. There are four dimensions to study: measurement length, applications, event trigger sources, and processor microarchitecture.

Our goal is to minimize measurement length to provide the best input resolution for super-resolution reconstruction. We will determine empirical values for the fixed and variable components of measurement input error and steady state error. Finally, we will evaluate performance counter error characteristics across a range of applications, event trigger sources, and processor microarchitectures.

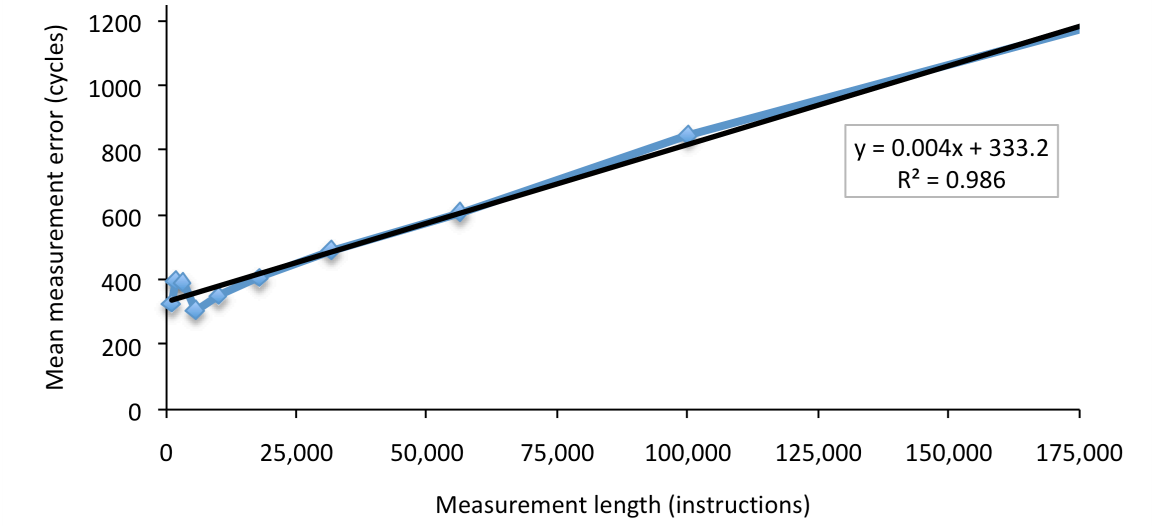


Figure 6.12: Performance counter error vs. measurement length

IntersetMO() with the cycle counter on a Core 2.

We selected one representative function and the one event source (cycles) since the cycle count is a summation of all other performance related events. Total error matches our hypothesis of constant average rate of steady state error plus a relatively stable amount of measurement interrupt error (fixed overhead). The regression equation tells us the steady state error rate (the error floor), and the size of the measurement interrupt error. We can also convert the R^2 value to a variance number for the measurement interrupt error.

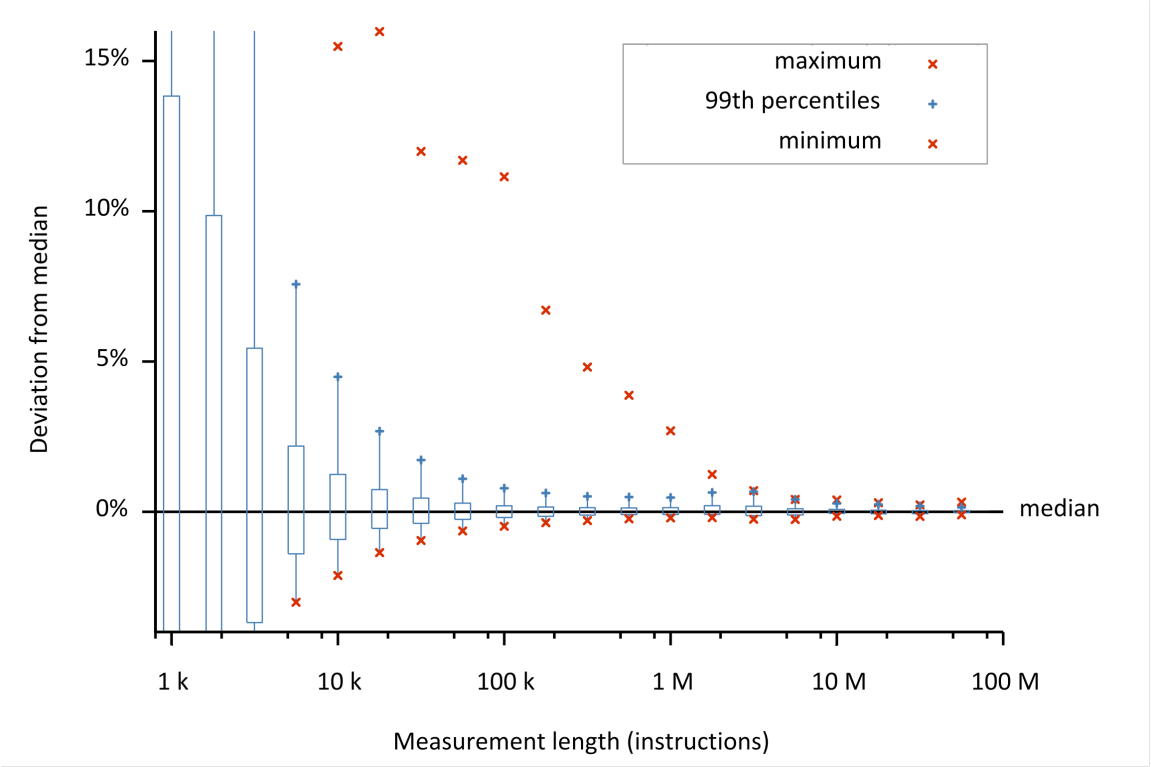


Figure 6.13: Performance counter error deviation from median

Next we look at the distribution of error at each point. In the previous charts we only looked at the mean error value. In this chart we use box plots, plus plots of the minimum and maximum errors to show the distribution characteristics of error. First, we see our hypothesis of a few large outliers is true. There is a big gap between the 99th percentile and the maximum value. Each measurement length has roughly the same distribution, with a cluster of points at the minimum value (note the exact overlap of the minimum and 99th percentile), and then a heavy positive tail.

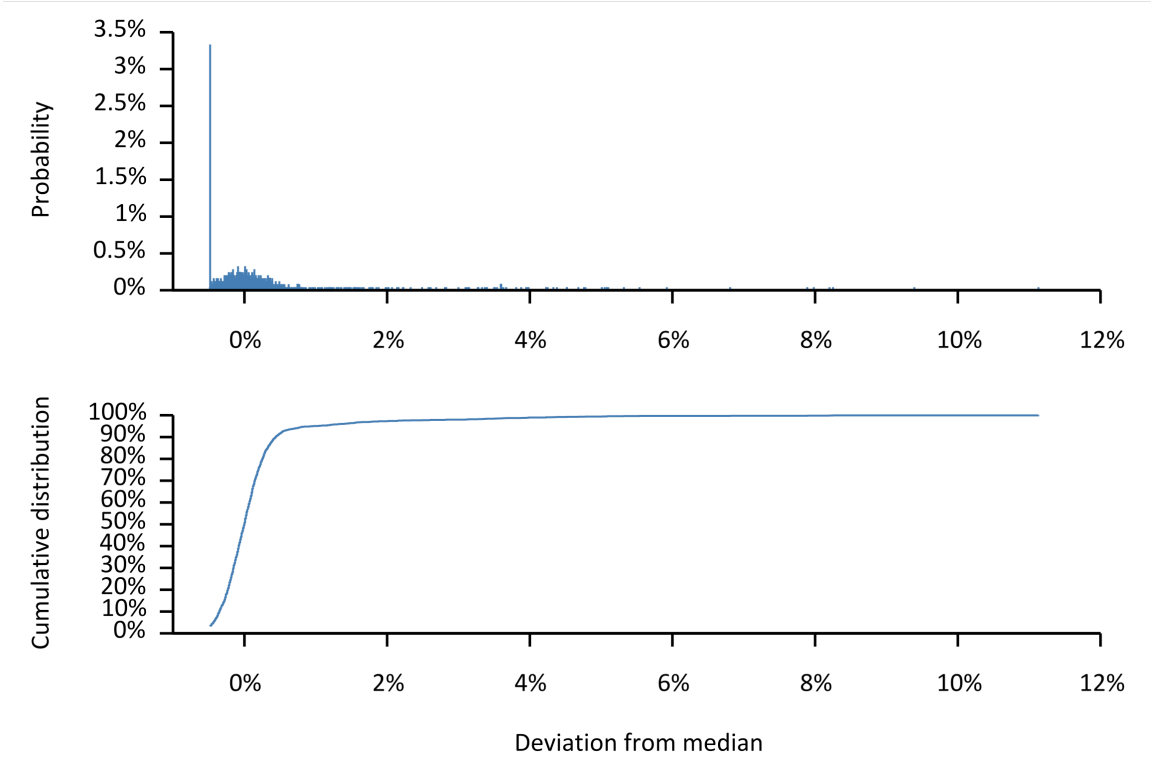


Figure 6.14: Distribution of error deviation from median

Finally, we take a look at the exact distribution of error for just one data point (100,000 instructions measurement length, in this case). We see a large concentration of measurements that match at a minimum value. Note that the measurements used in all the plots for section 2.5 are collected throughout the entire function, covering many different types of execution, and normalized to provide this convenient data set. Next, we see a normal distribution around the median, and a very heavy tail to the maximum. This portrays the same data as the previous plot. The 99th percentile is at about 1% deviation from the median, while the maximum is at 11%.

6.3.2 Initial reconstruction performance

We can verify the correct functioning of our super-resolution algorithms only if we know the 'correct' number of events for the small regions of execution that we reconstruct. We

can 'know' the correct values in two ways (1) by measuring at the same high resolution as the super-resolution reconstruction, or (2) by collecting a reference data set with emulation or simulation. The first approach of simply trying to measure at the same high resolution as our reconstruction will not work well because of the additional error in high-resolution measurements. This error is exactly what we hope to reduce with super-resolution. The second approach requires a cycle accurate simulator to produce the 'correct' number of microarchitectural events. However, functional instruction emulation is sufficient to validate super-resolution on architectural events such as instruction mix performance counters.

Purely synthetic data sets can also be used to tune our super-resolution algorithms. In addition, we can add controllable amounts of variability to the measurements to determine the error response of our algorithms. We have begun validation of our best to date super-resolution algorithms on synthetic data sets (sine wave and impulse functions) and real data collected on a Core 2 Duo running a SPIRAL [Püschel et al. 2005] generated 2^{15}-point, double precision, complex discreet Fourier transform. However, for brevity we present only our results on real data collected from the Core 2 Duo processor executing the 2^{15}-point DFT.

We collect a dynamic instruction trace from the same binary that we measure with the performance counters. The trace is collected by instrumenting the binary in memory using the Pin 2.0 for IA-32 binary instrumentation tool. We collect every PC that is executed, including system call code. We then lookup the assembly code of each PC from the disassembly of the binary. Finally, we classify each opcode to match the instruction mix performance counters for branches and SSE2 instructions, as well as memory load and store operations.

Table 6.5: Reconstruction sensitivity to number of measurements

2^{15}-point DFT super-resolution, error generally decreases with more measurements, but reconstruction runtime increases.

			Super-resolution reconstruction error		
Measurements	Ω	Magnification	Instructions retired	Branches retired	Reconstruction runtime
1000	40	6.9×	4.6%	3.5%	15 sec
500	40	6.9×	4.9%	2.6%	6.9 sec
200	40	6.9×	5.3%	2.1%	2.9 sec
75	40	6.9×	31%	16%	0.951 sec

Table 6.6: Reconstruction sensitivity to order of consecutive measurement constraints

Error generally decreases with higher order constraints, but reconstruction runtime increases.

			Super-resolution reconstruction error		
Measurements	Ω	Magnification	Instructions retired	Branches retired	Reconstruction runtime
500	100	6.9×	4.9%	2.5%	23 sec
500	40	6.9×	4.9%	2.6%	6.9 sec
500	20	6.9×	8.7%	3.2%	3.4 sec
500	2	6.9×	14%	6.5%	0.16 sec

Table 6.7: Reconstruction sensitivity to magnification

Error generally increases with higher magnification, and reconstruction runtime increases.

			Super-resolution reconstruction error		
Measurements	Ω	Magnification	Instructions retired	Branches retired	Reconstruction runtime
500	40	17.2×	13%	8.1%	22 sec
500	40	6.9×	4.9%	2.6%	6.9 sec
500	40	5.2×	3.2%	2.3%	5.3 sec
500	40	1.7×	3.0%	1.4%	1.8 sec

6.3.3 Super-resolution reconstruction performance

We present our initial results of our super-resolution reconstruction algorithm on the Core 2 platform while measuring the Intel IPP benchmark suite. Figure 6.15 illustrates the increasing perturbation introduced by performance counter interrupts. The effective resolution of approximately 100,000 instructions is evident from the mean error per measurement as compared to the median value for each measurement. A variable number

of repeated runs (minimum 30) were collected to compute the median measurement value with high confidence.

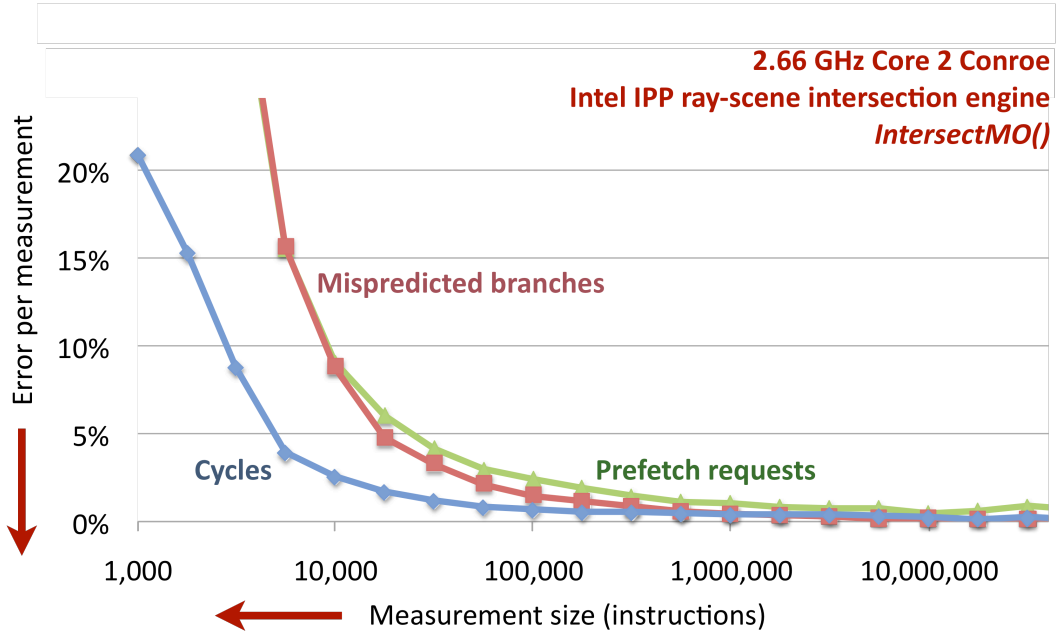


Figure 6.15: Performance counter resolution limits

Effective resolution limit is at approximately 100,000 instruction resolution.

Next, we investigate the sensitivity of reconstruction error to input measurement error. We introduced synthetic error that matched the empirical distributions measured to a performance trace of the IPP benchmark IntersectMO. Figure 6.16 shows the linear relationship between magnification and reconstruction error for multiple values of input measurement error.

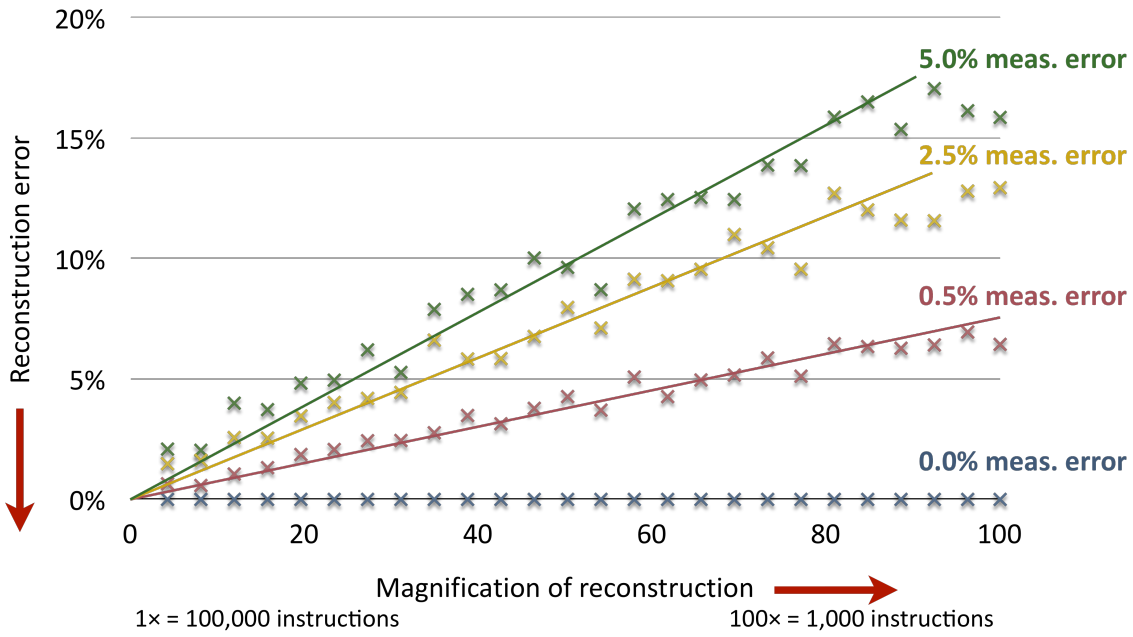


Figure 6.16: Super-resolution magnification sensitivity

Worse reconstruction results as input measurement error increases.

The slope of the linear correlation between magnification and reconstruction error is plotted in Figure 6.17 to show how we can increase the number of input program iterations to improve reconstruction error.

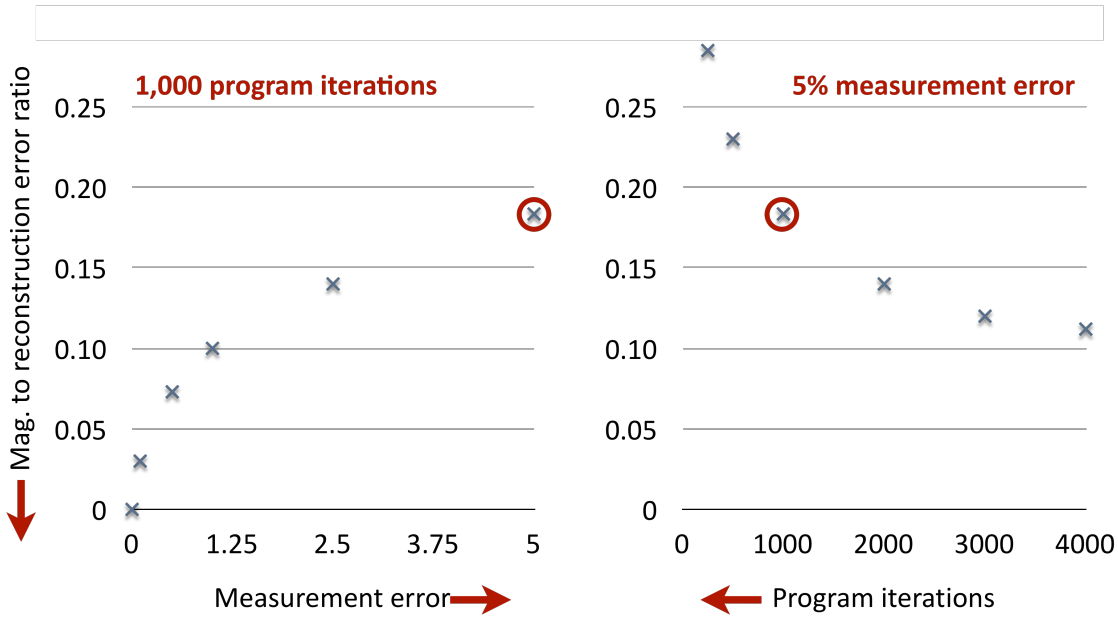


Figure 6.17: Super-resolution program iteration sensitivity

Better results with more iterations, but longer computation is required to perform the reconstruction.

Finally, the super-linear impact on the reconstruction runtime of increasing the input size, and thus the number of constraints, is plotted in Figure 6.18. Thus, while collection of data may not be a constraining factor, the runtime requirements of the reconstruction are usually the limiting factor in super-resolution.

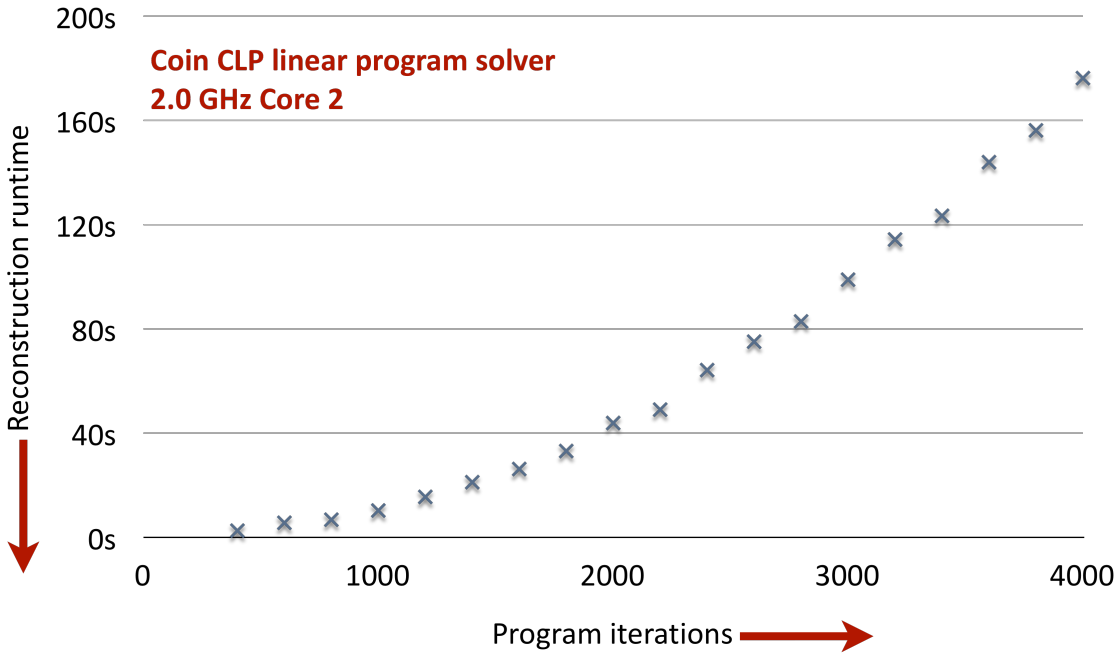


Figure 6.18: Super-resolution reconstruction runtime

There is significant future work in reducing the reconstruction runtime with LP refactoring. In addition, better model-fitted solutions than the bilateral total variation regularization may reduce reconstruction error. Finally, support for multi-threaded, non-deterministic programs would increase the applicability of this approach.

7 Conclusion

To address the need for improved simulation accuracy and performance, we propose the Sampling Microarchitecture Simulation (SMARTS) framework that applies statistical sampling to microarchitecture simulation. Unlike prior approaches to simulation sampling, SMARTS prescribes an exact and constructive procedure for sampling a minimal subset of a benchmark's instruction execution stream to estimate the performance of the complete benchmark with quantifiable confidence. The SMARTS procedure obviates the need for full-stream simulation by basing the strategy for optimal simulation sampling on the outcomes of fast sampling simulation runs.

We evaluated the SMARTS framework in the context of a wide-issue out-of-order superscalar simulator running the SPEC CPU2000 benchmark suite under two simulated processor configurations, as well as a full system 16-way multiprocessor simulation. SMARTSim, an implementation of SMARTS, is created by modifying SimpleScalar's sim-outorder to support systematic sampling. The results of our evaluations demonstrated the following: (1) SMARTSim achieves an actual average error of only 0.64% on CPI and 0.59% on EPI by simulating fewer than 50 million instructions in detail per benchmark. (2) By simulating exceedingly small fractions of complete benchmarks, SMARTSim achieves speed-ups of 35 and 60 times over full-stream simulation with sim-outorder for the two configurations.

The main performance bottleneck of SMARTS is the construction of accurate model state for each measurement by continuously warming large microarchitectural structures (e.g., caches and the branch predictor) while functionally simulating the billions of instructions

between measurements. This conservative approach, called functional warming, is applied to over 99% of each benchmark application. The speed of functional warming can be greatly accelerated using direct execution, or even hardware prototypes with limited detail except for the large, but simple, microarchitecture structures to be warmed.

We investigated replacing functional warming with checkpoints without sacrificing accuracy (we called this checkpointing framework TurboSMARTS). Checkpoints can be processed in random rather than program order, allowing simulation results and their statistical confidence to be reported while simulations are in progress. In addition, checkpoints can be processed independently, enabling parallel simulators that greatly reduce simulation latency.

SMARTS and TurboSMARTS offer accurate performance estimates (with a high quantifiable confidence) by taking a large number (e.g., 10,000) of short performance measurements over the full length of a benchmark. This simple random sampling does not exploit the often repetitive behaviors of benchmarks, collecting many similar, and possibly redundant, measurements. By identifying repetitive behaviors, we can apply stratified random sampling to achieve the same confidence as simple random sampling with far fewer measurements. We performed an oracle limit study of optimal stratified sampling of SPEC CPU2000 benchmarks to demonstrate that measurement can be reduced by 43 times over simple random sampling while matching result accuracy and confidence.

We evaluated three practical approaches for selecting strata, offline and online program phase detection, and IPC profiling. Program phase detection is attractive because it can be microarchitecture independent, while IPC profiling directly minimizes stratum variance, therefore minimizing sample size. Our results indicate that: (1) program phase detection falls far short of optimal opportunity, (2) IPC profiling requires expensive microarchitec-

ture-specific analysis, and (3) these methods require large sampling unit sizes to make strata selection feasible, offsetting their reductions of sample size.

The outcomes of this study have a fundamental bearing on simulator design. Designers should not attempt to accelerate detailed simulators at the cost of coding complexity or abstraction errors; instead designers should focus on increasing the simulator's flexibility and realism. For example, the SMARTS measurement framework has been successfully integrated into the Liberty Simulation Environment (LSE) by researchers at Princeton University [Penry, Vachharajani, and August 2005]. LSE is a computer architecture simulation infrastructure, which models microarchitecture at a structural rather than behavioral level of abstraction. As such, LSE models match hardware closely, but simulation is an order of magnitude slower than sim-outorder. Integration of SMARTS into LSE made typical simulation times tractable. The multiprocessor simulator, Flexus [Hardavellas et al. 2004], also employs the SMARTS measurement framework. Flexus implements a detailed microarchitecture model with sampling support to enable fast turnaround times on large server benchmarks.

The large speedup and parallelism enabled through statistical sampling and checkpointing have important implications for the design of future computer system simulators. First, the large speedups from sampling let simulator authors focus on designing flexible, modular simulators rather than optimizing for speed at all costs. Second, although hardware prototypes have many other advantages, it is not clear that they can reduce experiment turnaround time relative to sampled simulations. Finally, the 100 to 1,000 way parallelism available across measurements mitigates the need to multithread detailed simulation models.

While our opportunity study of stratified sampling shows promise for reducing sample size, our analysis of practical stratification techniques indicates little advantage over simple random sampling. Program phase detection stratification achieves only a small fraction of the available opportunity, since the discovered homogenous instruction footprints do not translate to homogenous performance. IPC profiling requires expensive and potentially non-portable stratification that is not justified by improvements in sample size. Neither approach improves in total measurement over simple random sampling because stratification cannot be performed at small sampling unit sizes. A promising approach that attempts to select low-variance strata is EXPERT [Liu and Huang 2004] that splits the instruction stream at large code loop and subroutine boundaries. Unfortunately, no comparison has yet been made with either SMARTS or SimPoint to determine EXPERT's relative performance in reducing sample size.

Thus, we conclude that future work remains for stratified sampling to provide performance improvements with statistical confidence intervals. In addition, our empirical efforts to minimize warming periods for microarchitectural state have room for improvement. More rigorous frameworks may be established to better define shorter warming periods and tighter error bounds.

Bibliography

AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. 1988. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. 6*, 4, 393–431.

ALAMELDEEN, A. R., WOOD, D. A. 2003. Variability in architectural simulations of multithreaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, Feb. 2003.

BAKER, S., AND KANADE, T. 2002. Limits on super-resolution and how to break them. *IEEE Trans. Pattern Anal. Mach. Intell. 24*, 9, 1167-1183.

BARR, K. C., PAN, H., ZHANG, M., AND ASANOVIC, K. 2005. Accelerating multiprocessor simulation with a memory timestamp record. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. Tech. Rep. 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

BURTSCHER, M. AND GANUSOV, I. 2004. Automatic synthesis of high-speed processor simulators. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

CAIN, H. W., LEPAK, K. M., SCHWARTZ, B. A., AND LIPASTI, M. H. 2002. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*, HPCA, Feb. 2002.

CALLISTER, J. 2006. The future of hardware performance monitors – What can we count on? In *Workshop on Functionality of Hardware Performance Monitors at MICRO-39*, Dec. 2006.

CHEN, S. 2004. Direct SMARTS: Accelerating microarchitectural simulation through direct execution. MS Thesis, Electrical and Computer Engineering, Carnegie Mellon University, June 2004.

CONTE, T. M., HIRSCH, M. A., AND MENEZES, K. N. 1996. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 14th International Conference on Computer Design*, Oct. 1996.

DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. 1997. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

DONGARRA, J., LONDON, K., MOORE, S., MUCCI, P., TERPSTRA, D., YOU, H. ET AL. 2003. Experiences and lessons learned with a portable interface to hardware performance counters. In *Proceedings of the Int. Parallel Distributed Process. Symp.*, Apr. 2003.

EASTON, M. C. AND FAGIN, R. 1978. Cold-start vs. warm-start miss ratios. *Commun. ACM 21*, 10, 866–872.

EECKHOUT, L., NUSSBAUM, S., SMITH, J. E., AND BOSSCHERE, K. D. 2003. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro 23*, 5, 26–38.

EECKHOUT, L., LUO, Y., DE BOSSCHERE, K., AND JOHN, L. K. 2005. BLRL: Accurate and efficient warmup for sampled processor simulation. *The Comput. Journal 48*, 4, 451–459.

EKMAN, M. AND STENSTRÖM, P. 2005. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the IEEE international Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

FARSIU, S., ROBINSON, D., ELAD, M., AND MILANFAR, P. 2003. Robust shift and add approach to super-resolution. In *Proceedings of the SPIE Conf. Applicati. Digital Image Process.*, Jan. 2003.

FARSIU, S., ROBINSON, D., ELAD, M., AND MILANFAR, P. 2004. Fast and robust multiframe super resolution. *IEEE Trans. Image Process. 13*, 10, 1327–1344.

FALCÓN, A., FARABOSCHI, P., AND ORTEGA, D. 2007. Combining simulation and virtualization through dynamic sampling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Apr. 2006.

FIELDS, B. A., BODIK, R., HILL, M. D., AND NEWBURN, C. J. 2004. Interaction cost and shotgun profiling. *ACM Trans. Architecture Code Optimization 1*, 3, 272–304.

GIRBAL, S., MOUCHARD, G., COHEN, A., AND TEMAM, O. 2003. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2003.

HAMERLY, G., PERELMAN, E., LAU, J., AND CALDER, B. 2005. SimPoint 3.0: Faster and more flexible program analysis. *Journal on Instruction-Level Parallelism*, Sept. 2005.

HARDAVELLAS, N., SOMOGYI, S., WENISCH, T. F., WUNDERLICH, R. E., CHEN, S., KIM, J., FALSAFI, B., HOE, J. C., AND NOWATZYK, A. G. 2004. SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev. 31*, 4, 31–34.

HASKINS, J. W. AND SKADRON, K. 2001. Minimal Subset Evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 19th International Conference on Computer Design*, Sept. 2001.

HASKINS, J. W. AND SKADRON, K. 2003. Memory Reference Reuse Latency: Accelerated warmup for sampled microarchitecture simulation. In *Proceedings of the International Symposium on the Performance Analysis of Systems and Software*, Mar. 2003.

HANKINS, R., DIEP, T., ANNAVARAM, M., HIRANO, B. ,ERI, H., NUECKEL, H., SHEN, J. P. 2003. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual IEEE/ACM international Symposium on Microarchitecture*, Dec. 2003.

HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. 1997. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, May 1997.

HENNING, J. L. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer 33*, 7, 28–35.

HILL, M. D., AND SMITH, A. J. 1989. Evaluating associativity in cpu caches. *IEEE Transactions on Computers 38*, 12, 1612–1630.

HSU, W. C., CHEN, H., AND YEW, P. C. 2002. On the predictability of program behavior using different input data sets. In *Workshop on Interaction between Compilers and Computer Architectures*, HPCA, Feb. 2002.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION 2002. ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). ISO/IEC 8825-1:2002 | ITU-T Rec. X.690.

IYENGAR, V. S., TREVILLYAN, L. H., AND BOSE, P. 1996. Representative traces for processor models with infinite cache. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*. Feb. 1996.

JAIN, R. K. 2001. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, NY.

KESSLER, R. E., HILL, M. D., AND WOOD, D. A. 1994. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput. 43*, 6, 664–675.

KORN, W., TELLER, P. J., AND CASTILLO, G. 2001. Just how accurate are performance counters? In *Proceedings of IEEE Int. Conf. Performance Computing Commun.*, Apr. 2001.

LAFAGE, T. AND SEZNEC, A. 2000. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *IEEE Workshop on Workload Characterization*, *ICCD*, Sept. 2000.

LAHA, S., PATEL, J. H., AND IYER, R. K. 1988. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput. 37*, 11, 1325–1336.

LAU, J., SAMPSON, J., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2005. The strong correlation between code signatures and performance. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

LAUTERBACH, G. 1994. Accelerating architectural simulation by parallel execution of trace samples. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, Jan. 1994, Vol. 1: Architecture, 205–210.

LEVY, P. S. AND LEMESHOW, S. 1999. *Sampling of Populations: Methods and Applications*. Wiley-Interscience, New York, NY.

LIU, W., AND HUANG, M. C. 2004. EXPERT: Expedited simulation exploiting program behavior repetition. In *Proceedings of the International Conference on Supercomputing*, Jun. 2004, 126–135.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., WERNER, B. 2002. Simics: A full system simulation platform, *Computer 35*, 2, 50–58.

MAXWELL, M. E., TELLER, P. J., SALAYANDIA, L., AND MOORE, S. 2002. Accuracy of performance monitoring hardware. In *Proceedings of the LACSI Symp.*, Oct. 2002.

PARK, S. C., PARK, M. K., AND KANG, M. G. 2003. Super-resolution image reconstruction: a technical overview. *IEEE Signal Processing Mag. 20*, 3, 21–36.

PELLEG, D. AND MOORE, A. 1999. Accelerating exact k-means algorithms with geometric reasoning. In *Proceedings of the Fifth ACM SIGKDD international Conference on Knowledge Discovery and Data Mining*, Aug. 1999.

PENRY, D. A., VACHHARAJANI, M., AND AUGUST, D. I. 2005. Rapid development of flexible validated processor models. In *Proceedings of the Workshop on Modeling, Benchmarking, and Simulation*, ISCA, Nov. 2005.

PERELMAN, E., HAMERLY, G., AND CALDER, B. 2003. Picking statistically valid and early simulation points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2003.

PHAM, T. Q., VAN VLIET, L. J., SCHUTTE, K. 2005. Influence of Signal-to-Noise Ratio and Point Spread Function on Limits of Super-Resolution. *Image Processing Algorithms and Systems 4*, 5672, 169–180.

PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B. ET AL. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE 93*, 2, 232–275.

REINHARDT, S. K., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., AND WOOD, D. A. 1993. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

SMITH, A. J. 1982. Cache memories. *ACM Comput. Surv. 14*, 3, 473–530.

SPRUNT, B. 2002. The basics of performance-monitoring hardware. *IEEE Micro 22*, 4, 64–71.

VAN BIESBROUCK, M., EECKHOUT, L., AND CALDER, B. 2005. Efficient sampling startup for sampled processor simulation. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, Nov. 2005.

WENISCH, T. F., WUNDERLICH, R. E., FASAFI, B., AND HOE, J. C. 2006. Simulation sampling with Live-points. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Mar. 2006.

WENISCH, T. F., WUNDERLICH, R. E., FERDMAN, M., AILAMAKI, A., FASAFI, B., AND HOE, J. C. 2006. SimFlex: Statistical Sampling of Computer System Simulation, *IEEE Micro Special Issue on Computer Architecture Simulation and Modeling 26,* 4, Jul. 2006.

WOOD, D. A., HILL, M. D., AND KESSLER, R. E. 1991. A model for estimating trace-sample miss ratios. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.

WUNDERLICH, R. E., WENISCH, T. F., FALSAFI, B., AND HOE, J. C. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.

WUNDERLICH, R. E., WENISCH, T. F., FALSAFI, B., AND HOE, J. C. 2004. An evaluation of stratified sampling of microarchitecture simulations. In *Third Annual Workshop on Duplicating, Deconstructing, and Debunking*, *ISCA*, June 2004.

WUNDERLICH, R. E., WENISCH, T. F., FALSAFI, B., AND HOE, J. C. 2006. Statistical sampling of microarchitecture simulation. *ACM Trans. Modeling and Computer Simulation 16*, 3, 197–224.

YI, J. J., KODAKARA, S. V., SENDAG, R., LILJA, D. J., AND HAWKINS, D. M. 2005. Characterizing and comparing prevailing simulation techniques. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Feb. 2005.