TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes

Thomas F. Wenisch Roland E. Wunderlich Babak Falsafi James C. Hoe
Computer Architecture Laboratory (CALCM)
Carnegie Mellon University, Pittsburgh, PA 15213-3890
{rolandw, twenisch, babak, jhoe}@ece.cmu.edu
http://www.ece.cmu.edu/~simflex

ABSTRACT

Recent research proposes accelerating processor microarchitecture simulation through statistical sampling. These proposals advocate detailed microarchitecture simulation of a large number (e.g., 10,000) of brief (e.g., 1000-instruction) execution windows to minimize instructions simulated and achieve high confidence in performance estimates. Unfortunately, correct measurement of such short execution windows requires highly accurate model state before each measurement. Prior techniques construct this state by continuously warming large microarchitectural structures (e.g., caches and the branch predictor) while emulating billions of instructions between measurements in an approach called functional warming. Although current sampling proposals require only minutes of detailed simulation, functional warming increases total turnaround time to hours.

In this paper we propose TurboSMARTS, a simulation framework that stores functionally-warmed state in a library of small, reusable checkpoints. Small checkpoints are desirable because of the need for thousands of measurements. However, it is not possible to precisely determine the accessed subset of state because of the speculative nature of modern microprocessors. TurboSMARTS replaces functional warming with checkpoints that store a minimal subset of warmed state, reducing simulation runtime to minutes. We demonstrate that TurboSMARTS is more accurate and faster than alternatives that only checkpoint architectural state. TurboSMARTS can match the accuracy of prior techniques (i.e., ±3% error with virtual certainty), while estimating the performance of an 8-way out-of-order superscalar processor running SPEC CPU2000 in 91 seconds per benchmark, on average, with a 12 GB checkpoint library for the entire benchmark suite.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: *Measurement techniques, Modeling techniques*; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Measurement, Performance, Design

Keywords

Checkpointed microarchitecture simulation, simulation sampling, cold-start bias, SPEC CPU2000 simulation

1. INTRODUCTION

Computer architecture research routinely uses detailed cycle-accurate simulation to explore and validate microarchitectural innovations. Ideally, simulation studies should use the same benchmarks used to assess real hardware. Unfortunately, benchmark applications that are tuned to run for minutes on real hardware can require over a month to execute on today's high performance microarchitecture simulators [2,3,9,27,30].

Past research advocates sampling [1,5,20,21,32,34]—i.e., measuring only a subset of benchmark execution—as a technique to accelerate microarchitecture simulation. Many such studies advocate systematic sampling [5,21,34] using rigorous statistical theory to provide explicit validation that the measured portions accurately represent the behavior of a benchmark. Sampling theory dictates that, for a given accuracy, the number of measurements required for estimates depends only on the variability of the target metric and is independent of benchmark length [16]. Wunderlich et al., [34] determined that a sampling simulator can minimize instructions simulated by collecting a large number (e.g., 10,000) of brief (e.g., 1000-instruction) simulation windows.

A simulated microarchitecture's state must be accurately warmed prior to each of these sample measurements to produce unbiased estimates. Existing techniques achieve the lowest error by continuously warming large microarchitectural structures (e.g., caches and the branch predictor) while emulating the billions of instructions between measurements. This state updating process, referred to as functional warming [34], dominates simulation turnaround time because the entire benchmark's execution must be emulated, even though only a tiny fraction of the execution is simulated using detailed microarchitecture timing models. Thus, accelerating functional warming directly improves simulation sampling speed.

A replacement for functional warming must provide accurate warmed state and rapidly move to an execution trace location. In this paper we present a simulation framework, TurboSMARTS, that stores functionally-warmed state in checkpoints reusable across varied microarchitecture configurations. This technique eliminates functional warming by precomputing and storing a minimal subset of warmed state in a library of checkpoints for use in subsequent simulations. TurboSMARTS is based on two key observations: (1) only a small amount of architectural and microarchitectural state is accessed during the brief measurement periods of simulation sampling, and (2) checkpointing functionally-warmed state to perform these measurements is a more accurate method than alternatives that only checkpoint architectural state.

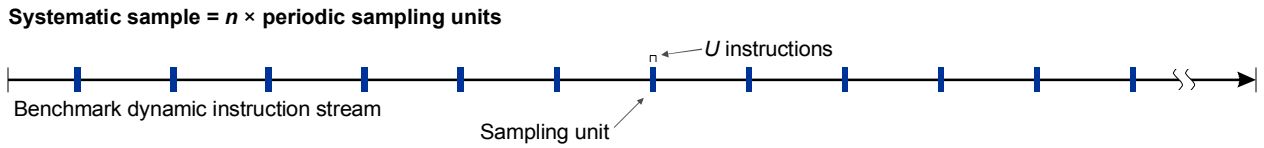

Figure 1. Systematic sampling variables for simulation sampling.

We present results from a TurboSMARTS-based simulator derived from SimpleScalar 3.0 `sim-outorder` [3] simulating the execution of the SPEC CPU2000 (SPEC2K) benchmarks on two microarchitecture configurations to show:

- **Benchmark simulation in minutes.** TurboSMARTS achieves an average simulation time for an 8-way out-of-order superscalar of just 91 seconds per benchmark, while bounding estimated CPI error to within +/-3% with virtual certainty. In contrast, an equivalent sampling simulator with functional warming and identical accuracy and confidence in results takes 7.0 hours per benchmark on average. Moreover, TurboSMARTS' runtime is independent of benchmark length, and instead depends only on a target metric's variability and the speed of detailed simulation.
- **Small, accurate, and reusable checkpoints.** While the required subset of model state for each simulation window is small, the precise subset is unknown *a priori* because of the speculative nature of execution in modern microprocessors. We show that whereas wrong-path (speculative) instruction *latency* affects scheduling through pipeline resource contention, wrong-path operand *values* rarely affect instruction throughput. We therefore advocate a checkpointing solution in which only correct-path data values in memory and the cache hierarchy are stored in each checkpoint. Our solution introduces no additional error over existing simulation sampling techniques. A complete library of checkpoints for SPEC2K requires 12 GB, as opposed to naïve checkpointing of all architectural state, which requires 7.2 TB of storage.
- **Leveraging of independent checkpoints.** Our design of checkpoints with no restrictions on processing order allows for parallelization, and prevents simulation of excess checkpoints. By processing checkpoints in a random order, we can terminate simulations precisely when a desired statistical confidence level is reached. In contrast, simulators that use functional warming require a strict program-order simulation to allow for unbiased sampling, preventing parallelization and runtime changes to the number of measurements.

This paper is organized as follows. In Section 2, we present background material on sample design and modes of simulation for simulation sampling. In Section 3, we evaluate warming methods for accuracy, flexibility, and speed. We detail the design of TurboSMARTS checkpoints in Section 4, and present the complete TurboSMARTS framework in Section 5. Section 6 presents performance results and analysis. Related work is described in Section 7. We conclude in Section 8.

2. BACKGROUND

Wunderlich et al. [34] advocate measuring many small regions of a benchmark to estimate performance metrics (e.g., CPI) with both high accuracy and high statistical confidence, while minimizing the number of instructions measured. However, the functional warming required by the proposed framework, SMARTS, dominates simulation runtime (and becomes a larger performance bottleneck with ever longer benchmark programs). In this section we present the sampling concepts needed to understand the sample design that minimizes measurement. We then present the modes of simulation used by an implementation of this sample design.

2.1 Sample design

Inferential statistics provide methods to estimate parameters of a large population from a representative subset [16]. We are interested in the estimation of performance (CPI, power, etc.) of benchmark programs on a simulated microarchitecture. Thus, we define the dynamic instruction stream as the population to be sampled.

The most basic sample design is simple random sampling, where every member of a population has an equal probability of being included in the *sample*. Figure 1 illustrates an alternative to random sampling, *systematic sampling*, where measurements are taken at periodic intervals. Systematic sampling can be simpler to implement than random sampling and has the same accuracy, except in the unlikely case that behaviors recur with precisely the same periodicity as measurements. Sampling theory calls the individual performance measurements collected over the course of a benchmark *sampling units*. The number of units in the sample is called the *sample size* (n). Each sampling unit measures the performance of a fixed number of instructions, we call this the *sampling unit size* (U).

There are two components of any error present in sampling estimates: *sampling error* and *non-sampling error*. Sampling error results when, by chance, poor measurement locations are chosen. Non-sampling error is caused by bias in measurement, for example a cold-start bias from empty caches at the start of each sampling unit producing incorrectly low performance estimates.

The probabilistic range of sampling error, called the confidence interval, is narrowed by increasing the sample size. Statistical sampling theory shows that the sample size required for a desired confidence interval depends exclusively on the variability of the estimated metric.

The study performed by Wunderlich et al. [34] investigated the variability of CPI for SPEC2K across a range of sampling unit sizes. Thus, they were able to determine the optimal sample design that minimized measured instructions. The results indicate that taking many small measurements allows for the least detailed simulation to achieve a desired confidence interval.

The optimal sample design recommendations of the SMARTS framework were n = 10,000 and U = 1,000 instructions. Using this sample design, it was shown that a SMARTS-based simulator could typically estimate an out-of-order superscalar's performance to

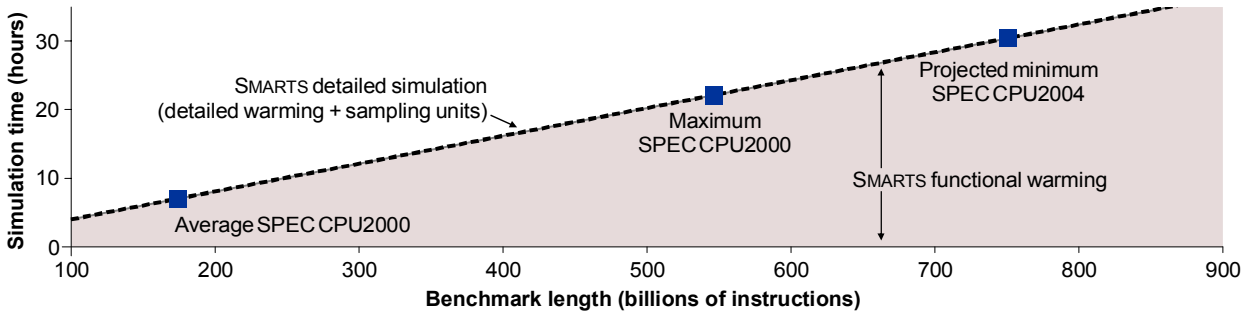

Figure 2. Breakdown of typical SMARTS execution. Detailed simulation takes only a few minutes, while functional warming takes hours.

better than ±3% error with virtual certainty for SPEC2K benchmarks. We use this same sample design to evaluate checkpointed simulation sampling. Thus, all of the simulation sampling methods evaluated in Section 3 have identical confidence intervals, and only have differences in non-sampling error.

2.2 Modes of simulation

The primary challenge of simulation sampling is minimizing non-sampling error. The most common cause of this is the cold-start bias of unwarmed microarchitectural structures. SMARTS allows for bias-free performance estimates from very small sampling units by constructing accurate initial architectural (e.g., register and memory values) and microarchitectural (e.g., pipeline components and the cache hierarchy) state before collecting each measurement.

SMARTS uses a two-tiered strategy to construct a sampling unit's initial state. Prior to each measurement, microarchitectural structures whose current state reflects the history of a small, bounded set of recent instructions—such as the reorder buffer or issue queue—are warmed through *detailed warming*, brief simulation (e.g., a few thousand instructions) of the complete detailed performance model sufficient to warm such small structures.

The second component of the SMARTS warming strategy, *functional warming*, addresses state updates between two measurements. Like other simulation sampling frameworks [1,20,23,32], SMARTS emulates each instruction to update architectural state. To minimize and bound detailed warming requirements, SMARTS continuously updates the remaining few microarchitectural structures—caches, TLBs, and branch predictors. These structures have state that persists across measurements, and they cannot be warmed sufficiently by a brief detailed warming period.

Unfortunately, functional warming, as proposed, is a performance bottleneck in simulation sampling [11,34]. Given typical cycle-accurate simulation models for microprocessors (e.g., SimpleScalar's `sim-outorder` [3]), the performance measurement of a wide-issue out-of-order superscalar using the SMARTS sample design requires little detailed simulation: as small as a few seconds, and only a few minutes on average on a state-of-the-art host machine. A SMARTS-based simulator's total runtime, however, is orders of magnitude longer than the detailed simulation time because runtime is dominated by the functional warming between measurements.

The trend towards longer-running benchmarks in future suites exacerbates the functional warming bottleneck. The sample size, and thus the detailed simulation time, only depend on a processor's performance variability across a benchmark's execution (which does not change drastically across benchmarks or microarchitectures) [23,34]. However, functional warming time is directly proportional to benchmark length because the entire execution must be simulated. As such, functional warming requirements will continue to grow as we lengthen benchmarks to scale with hardware performance improvement [33]. Figure 2 depicts this time breakdown and extrapolates the simulation time required for the shortest possible SPEC CPU2004 benchmarks based on the SPEC submission requirements.

3. WARMING WITH CHECKPOINTS

The computation performed during functional warming is often similar or identical across simulations of the same benchmark. Thus, we may alleviate the functional warming bottleneck by checkpointing simulation state. We memoize the redundant calculation of functional warming across runs, amortizing the one-time cost of computing warmed state.

For each portion of model state, we choose either to store the state in checkpoints, or dynamically construct the state via functional or detailed warming after a checkpoint is loaded. This choice impacts simulation sampling along three dimensions: the accuracy of the warmed state, the reusability of checkpoints across simulator configurations, and the speed of simulation. This section explores the design space of warming methods with respect to these three dimensions.

3.1 Simulation sampling warming methods

There is a rich design space of possible warming strategies that combine checkpointing and dynamic warming for various portions of microarchitectural model state. We evaluate this space in the context of the sample design that gives the shortest possible detailed simulation time for a chosen target sampling error: a large sample of small sampling units.

We employ detailed warming to initialize queue and pipeline state in all of the warming strategies we consider. Detailed warming can reconstruct state for the vast majority of microarchitectural structures rapidly, and the amount of required warmup can be determined via worst-case analysis [34]. By warming most structures

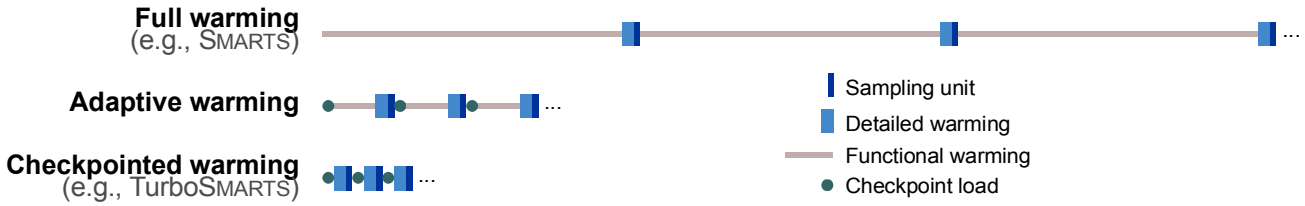

Figure 3. Simulation sampling warming methods. All methods use the same sample design and confidence intervals, only non-sampling error differs.

dynamically, we avoid storing any state for these structures, and do not fix 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 non-sampling error introduced by the warming strategy. We use the SMARTS sample design to give low sampling error with a minimum of measurement, as described in Section 2.1. SMARTS demonstrated very low non-sampling error—0.6% on average, 1.6% worst case [34]—using functional warming. It is essential to maintain this high accuracy when accelerating warming because statistical techniques, such as confidence intervals, cannot directly assess non-sampling error. We must rely on a sound warming strategy to ensure that non-sampling error is negligible.

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 (i.e., the maximum size or associativity of a cache as described in Section 4.2).

Finally, we evaluate the speed of warming alternatives in two ways. First, we consider how fast measurements can be processed. For all alternatives, detailed warmup and measurement time is the same, while functional warming and checkpoint decompression/loading time varies. Second, we consider whether each measurement is independent and can therefore be performed in parallel.

Warming methods. Figure 3 depicts alternatives in the warming strategy design space. At one extreme, functional warming is used for the entire duration between measurements, without checkpoints. 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. Checkpointed warming still uses detailed warmup for pipeline registers and queues, and does not limit configuration parameters for these structures.

Finally, we can choose to warm some structures, but not others. The most logical *adaptive warming* approach is to checkpoint architectural state, which does not vary across simulations, while using functional warming to regenerate microarchitectural state that varies across experiments (caches, branch predictor). Simulating workloads for which architectural state does vary across repeated runs—i.e., because of interrupt timing or different interleaving of multiprocessor instruction streams—is beyond the scope of this work. Adaptive warming can save a significant fraction of time over full warming, but requires us to determine how much functional warming is necessary for a particular combination of workload and microarchitectural structures.

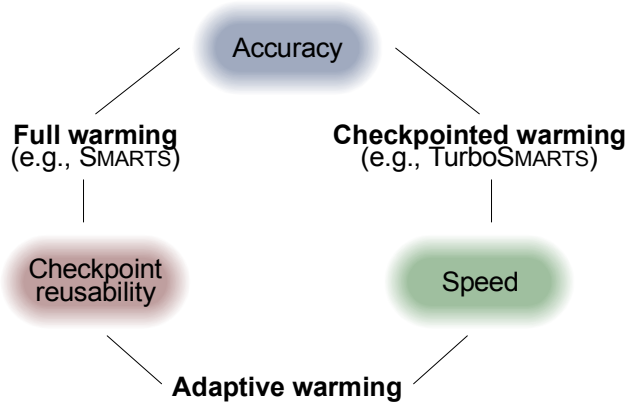

Figure 4. Relative merits of warming methods.

Trade-offs. Figure 4 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.

Full warming maximizes accuracy and flexibility, but its need for long periods of functional warming makes it slow. Full warming is explored and optimized in SMARTS. The high accuracy achievable with functional warming of caches and branch predictors has been demonstrated in [34] and independently confirmed in [36]. 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. Adaptive warming introduces the problem of determining how much functional warming a particular benchmark and microarchitecture configuration requires. The accuracy and speed of adaptive warming depend on a rigorous answer to this question. Unfortunately, determining the correct amount of warming is a difficult and unsovled problem [18].

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 warming alternative. Furthermore, it is straightforward to create checkpoints that are independent of

Table 1. Microarchitecture configurations.

Parameter	8-way (baseline)	16-way
RUU/LSQ size	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 4M 8-way L2 32-entry store buffer
L1/L2 line size	32/128 bytes	32/128 bytes
L1/L2/mem latency	1/12/100 cycles	2/16/100 cycles
ITLB/DTLB	4-way 128 entries/ 4-way 256 entries 200 cycle miss	4-way 128 entries/ 4-way 256 entries 200 cycle miss
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
Detailed warmup	2000 instructions	4000 instructions

one another. This allows checkpoint processing to be parallelized, reducing simulation latency. The drawback of checkpointed warming is that it imposes limits on some aspects of the simulated microarchitecture parameters, which constrains checkpoint reusability. Reusability is important because we must amortize the one-time cost of checkpoint creation (the cost is roughly the same as a full warming simulation) over many experiments.

In Section 3.3, we evaluate an adaptive warming approach and conclude that current methods for estimating cache warmup requirements do not meet our tight accuracy demands. In Section 4, we design a checkpointing approach that minimizes checkpoint storage cost and processing time without sacrificing any accuracy, while placing as few constraints as possible on simulator configuration. We solve the key problem of storing only the minimal subset of simulation state, without introducing bias.

3.2 Methodology

We evaluate TurboSMARTS and compare it to other warming alternatives using a sampling simulator based on the SimpleScalar 3.0 `sim-outorder` simulator [3] for the Alpha ISA. Without loss of generality, we use CPI (cycles-per-instruction) as our performance metric. Simulation sampling, however, has been shown to be applicable to other performance metrics of choice [34]. For improved realism, we modified the memory subsystem to include a store buffer and miss status holding registers (MSHRs), and model interconnect bottlenecks in the memory hierarchy. Checkpoints are encoded using ASN.1 DER format [15] and compressed with `gzip`, which incur minimal storage and processing time overhead. We use all 26 SPEC2K benchmarks [13] and 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.

We evaluate two microarchitecture configurations in this study. 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 datapath, 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 1.

Our experiments were performed on systems with dual 2.80 GHz Intel Xeon (512 KB L2) processors and 3 GB of PC2700 DRAM. All simulator runtime results were collected with only a single simulation running per system, and with Hyper-Threading disabled. We stored checkpoint libraries on 250 GB Hitachi Deskstar 7K250 hard drives, which have a mean sequential data read rate of 45 MB/s.

For the adaptive warming results in Section 3.3 we use MRRL 0.0.1 [12] to determine appropriate warmup periods for each measurement location.

3.3 Adaptive warming evaluation

The key challenge of adaptive warming is determining the warming period length. If this length is underestimated, simulation results will be biased. If this length is overestimated, we sacrifice simulation speed.

A recently proposed technique for determining cache warmup requirement is Memory Reference Reuse Latency (MRRL) [12]. MRRL collects a histogram of memory access reuse distances between each pair of measurement locations during a functional simulation of a benchmark. The warmup length reported by MRRL is the length sufficient to cover 99.9% of the observed reuse distances. This bound on cache warmup requirements is configuration independent, since reuse latency is measured by instruction count in a functional simulator. This analysis must be run once per benchmark and takes roughly the same time as a single full-warmup simulation run.

MRRL has demonstrated low non-sampling error on large sampling units (worst case error of 2% for 50-million-instruction sampling units). However, MRRL has not been evaluated on the small sampling units required by the optimal sample design. Small sampling units are more susceptible to bias because warming errors cannot be amortized over a large sampling unit. Measuring longer units reduces sensitivity to accurate warmup periods because early portions of the measurement effectively perform additional warming for later portions.

We evaluate MRRL with a reuse probability of 99.9% as recommended by the MRRL authors. This reuse probability results in an average of 4.1 million instructions of warmup prior to each sampling unit, which is 20% of the average full warming interval (20.5 million instructions). Thus, an approximation of the runtime of this 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).

We present the results of our evaluation of adaptive warming with MRRL for small units in Figure 5. Both average (1.1%) and worst-case error (5.4%) are considerably worse than full warming. Error

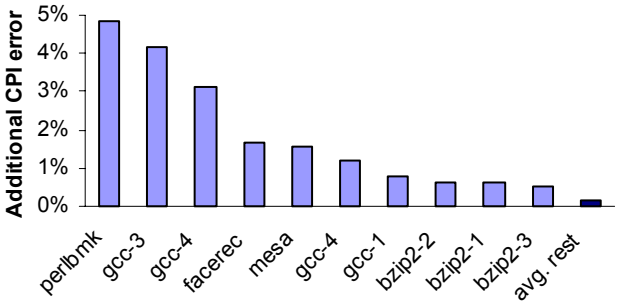

Figure 5. Additional error incurred by adaptive warming using MRRL vs. full warming.

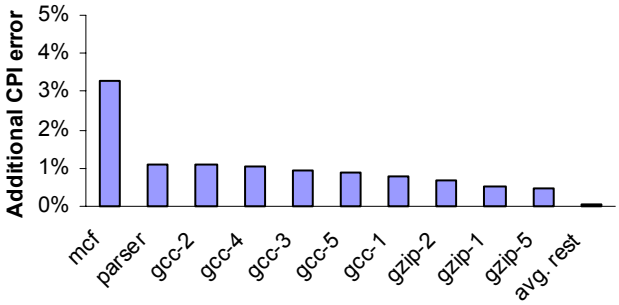

Figure 6. Additional error incurred when ignoring wrong-path with checkpoints of only correct-path accessed state.

is high because short measurements are very sensitive to accurate cache state.

MRRL does not allow for independent checkpoints, since cache state must be stitched [18] from previous measurements. This precludes simple parallelization of the checkpointed simulation, and also prevents random order processing of the checkpoints, the advantages of which are expanded upon in Section 5.1. If MRRL is used without stitched state (this assumes an empty cache upon the load of each checkpoint) we observe a CPI error increase to an average of 1.9%, with a worst case of 11%.

From this analysis, we conclude that currently-known warmup analysis techniques cannot meet our accuracy goal, while providing a speed advantage over full warming. Increasing warming over MRRL (or increasing the MRRL reuse probability threshold) improves accuracy, but further reduces the speed of adaptive warming. In the rest of this paper, we focus on checkpointed warming. We design a checkpointing approach that maximizes speed, without losing any accuracy and sacrifices as little checkpoint reusability as possible.

4. CHECKPOINT DESIGN

The goal of TurboSMARTS is to improve simulation speed over prior full warming approaches without sacrificing accuracy or confidence in results. In Section 3.3 we explored adaptive warming with MRRL to determine warmup length, and found that this approach does not meet our accuracy goal. Thus, we turn our investigation to checkpointed warming. Because our sample design requires an average of 8000 measurements per benchmark (not n = 10,000 as in SMARTS, due to random order processing; see Section 5.1), minimizing checkpoint size is a paramount concern. If typical compressed checkpoint size exceeds 1 MB, we will be unable to store a complete checkpoint library for SPEC2K on a single high-capacity disk (400 GB).

We begin our checkpoint design process by assuming fixed cache, TLB, and branch predictor table organizations (fixed capacity, associativity, block size, etc.). Note that we do not fix latency parameters for these components, only the parameters which affect structure contents. In Section 4.1, we identify the minimal state that must be checkpointed to obtain accurate results assuming this fixed configuration. In Section 4.2, we provide techniques for relaxing these constraints and improving the reusability of checkpoints with little additional cost in space, creation time, and processing time.

4.1 Minimizing checkpoint size

Identical accuracy to full warming simulation can be achieved by storing all architectural and functionally-warmed microarchitectural state in checkpoints. When all state is checkpointed, the initial simulation state for each measurement will be the same as full warming, and the checkpointed simulation reproduce identical execution traces.

Unfortunately, checkpointing all state is intractable. SPEC2K benchmarks are designed to have memory footprints up to 200MB [13]. We measured an average memory footprint of 105 MB. Checkpoints containing all architectural state would require 33 TB of storage (uncompressed) for 8000 measurements for all benchmark/input combinations of SPEC2K. Based on the typical `gzip` compression we observed, we estimate a total storage cost of 7.2 TB after compression. At this size, simulations would be I/O bound, and provide little, if any, speedup over full warming simulation. It may be possible to save space by storing only changes to memory between checkpoints, but this approach sacrifices checkpoint independence.

Instead, we take advantage of the short length of our measurement and detailed warming periods. Simulation state that is never referenced during measurement or detailed warming can be omitted from the checkpoint without affecting the simulation. Since the detailed simulation periods are just a few thousand instructions, only a tiny subset of state is accessed. Our design seeks to identify and store only the minimal set of accessed state.

We can identify precisely which instructions will commit during warmup and measurement when we create checkpoints. Thus, it is straightforward to identify all the architectural registers, memory locations, cache data, cache tags, and branch predictor state these instructions will access—generally less than 32 KB per checkpoint (uncompressed, including the ASN.1 overhead).

However, this approach 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 checkpoint creation time. To do so requires either fixing the vast majority of simulation parameters (queue sizes and latencies), or requires exploring all possible speculative paths to the depth they might be followed (as bounded by, for example, ROB size). The former virtually eliminates any checkpoint reusability, while the latter requires analysis that grows exponentially with speculation depth.

Although the effects of wrong-path instructions on the commit instruction stream are generally small [4], they cannot be ignored given our tight non-sampling error goals. Errors in wrong-path modeling cause the schedule of wrong-path execution to differ from a simulation where all accessed state is available. These schedule differences impact the execution of correct-path instructions indirectly through functional unit and memory queue/bandwidth contention.

We measured the non-sampling error introduced by checkpointing only the state accessed on the correct-path. Whenever any wrong-path instruction accessed unavailable state, we provide a default outcome to approximate the most common case (e.g., assume that all wrong-path loads hit, provide a zero for loads to unavailable memory, etc.). While the average non-sampling error increase for CPI is only 0.1%, the worst case is 3.3%. Figure 6 shows the non-sampling error results for the benchmarks with the most error.

We must properly model the resource contention of wrong-path instructions to eliminate this error. The sequence of wrong-path instructions which enter the microarchitecture is largely determined by the branch predictor, as it directs wrong-path fetch. The schedule of execution is determined by the decoded instruction stream. An important exception is wrong-path load instructions, whose resource utilization depends on whether or not the load hits in the cache hierarchy. Loads which hit trigger the execution of dependant instructions. Loads which miss occupy cache and memory bus bandwidth and queuing resources.

These properties of instruction scheduling suggest a checkpointing approach where we store the complete state of the branch predictor and cache tag arrays, but omit memory values unless they are accessed on the correct-path. If a wrong-path load accesses a memory location for which the value is unavailable, but the cache indicates a hit, we return a zero value as in our correct-path only experiment. This "poisoned" value does not introduce error into the wrong-path execution schedule unless it propagates to a dependant branch or address calculation. Such poison propagation events are relatively rare, occurring significantly less than once per checkpoint, on average, and do not measurably affect non-sampling error.

By checkpointing only cache tags, branch predictor state, and correct-path data values, TurboSMARTS achieves non-sampling error matching that of full warming simulation. This approach allows TurboSMARTS to omit the vast majority of memory state from checkpoints, requiring, on average, only 142 KB of state per checkpoint (uncompressed).

4.2 Maximizing checkpoint reusability

In Section 4.1, we assumed fixed cache, TLB, and branch predictor organizations when storing checkpoints. To improve checkpoint reusability, we aim to relax these constraints as much as possible. We maximize reusability to amortize the runtime cost of creating a checkpoint library over as many experiments as possible. This eliminates the need to create new checkpoints each time one of these structures is changed. Furthermore, we minimize the storage cost of supporting a wide range of configurations.

There are two basic approaches to increasing checkpoint reusability. First, we can collect state snapshots for multiple component configurations in a single checkpoint creation pass. The second, preferable approach, is to modify the saved representation of component state such that a range of organizations can be reconstructed when a checkpoint is loaded. However, it is difficult to 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 straightforward and effective if the number of configurations is relatively small. The major cost of checkpoint creation is the traversal of the entire benchmark instruction stream, as in full warming simulation. 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 checkpointed state for both configurations in a single pass. We use 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 that can accurately reconstruct a range of configurations. This idea has been widely applied in the context of trace-based cache research [14]. Prior research on trace-based cache simulation has shown that, for many typical replacement algorithms, alternative cache organizations exhibit inclusion and set refinement [14]. Inclusion is the property that larger caches (more sets or greater associativity, but fixed block size) contain a superset of the contents of smaller caches. Set refinement holds that two blocks that map to the same set in a larger cache also map to the same set in a smaller cache. By recording a trace of accesses that constructs the correct tag state for a particular cache organization, we can exploit these properties to use this same trace to reconstruct all smaller and less associative caches. We refer to this representation of cache tag arrays as a *tag trace*.

To generate a tag trace, we determine the total ordering of the last access times of all tags in the cache. By sorting tags according to this total order, the result is an access trace that can be replayed into the cache to reconstruct a correct tag image. We obtain the total ordering by recording timestamps of each cache access with the accessed tag array entry throughout checkpoint generation.

The tag trace technique allows a particular checkpoint library to be reused when simulating any smaller or lower associativity cache, provided we do not alter the cache block size or replacement policy. By combining this modified representation for caches with multiple stored branch predictor configurations, we can create a single checkpoint library that can simulate the cross product of configurations, using a single checkpoint creation pass, and without any redundant storage.

5. TURBOSMARTS FRAMEWORK

In this section, we present issues that arise when using TurboSMARTS' checkpointed warming in practice. Section 5.1 explains how to take advantage of independent checkpoints to parallelize simulation and avoid processing excess checkpoints. Section 5.2 addresses issues in sizing a checkpoint library to avoid frequently falling short of target confidence when a library is too

small. Finally, Section 5.3 summarizes a unified procedure for experimentation with TurboSMARTS.

5.1 Leveraging independent checkpoints

One key advantage of TurboSMARTS over other warming techniques is that each TurboSMARTS checkpoint is independent, and can thus be processed in isolation. Independent checkpoints allow simulation latency to be reduced arbitrarily by parallelizing checkpoint processing across multiple CPUs. In the limit, each checkpoint could be processed on its own CPU, reducing simulation latency to milliseconds.

In contrast, both full warming and adaptive warming require that measurements be processed in program order. With full warming, each simulation must dynamically generate architectural state, requiring emulation from the beginning of a benchmark. With adaptive warming, non-sampling error increases sharply if cache state is not stitched from the end of one measurement to the start of the next, as shown in Section 3.3.

Independent checkpoints also provide flexibility in sample design. When checkpoints are processed in a random order, a simple random sample has been taken after any number of measurements have been completed. For example, if there are 10,000 checkpoints taken at periodic intervals, after 500 random checkpoints have been processed we have evaluated a random sample of size 500. We can calculate a confidence interval from the observed variance in measurement, and if the interval is acceptable, simulation can stop. Otherwise, we process additional random checkpoints until the desired confidence level is reached. Instead, if sampling units are measured in program order as in SMARTS, all sampling units must be measured before any performance results can be reported.

In TurboSMARTS, we advocate shuffling the available checkpoint library into a random order prior to simulation. By using this optimization, TurboSMARTS processes only the required sample size for the observed variance of each experiment, and does not waste time processing excess checkpoints.

5.2 Sizing a checkpoint library

When creating a checkpoint library, we must choose the number of checkpoints to include in the sample of each benchmark. Random shuffling of the checkpoint library allows us to avoid processing excess checkpoints if the library is too large. However, experiments will fail to meet their target confidence intervals if a checkpoint library is too small.

As described in [34], the sample size necessary to achieve a desired confidence interval depends solely on the coefficient of variation, V_x, of the target metric. When V_x is known (for example, because an experiment is repeated), it is straightforward to calculating the minimal sample size that will achieve the desired confidence interval.

However, for a new combination of microarchitecture configuration and benchmark, it is difficult to predict the new V_x. For example, in our 16-way configuration, instructions can commit twice as quickly as in our 8-way configuration when available instruction level parallelism is high, while both microarchitectures will be dataflow-bound when parallelism is low. Because performance of the 16-way configuration can vary over a wider range, we might expect V_{CPI} to be universally higher than in the 8-way configuration. Contrary to this intuition, V_{CPI} of 16-way is lower for some benchmarks. Figure 7 graphs the change in V_{CPI} for the subset of our benchmarks that exhibit the largest variation in each direction. The plotted subset shows that the same microarchitectural changes can significantly affect V_{CPI} in either direction. Since sample size is quadratically related to V_{CPI}, unpredictable changes in V_{CPI} can lead to large swings in required sample size.

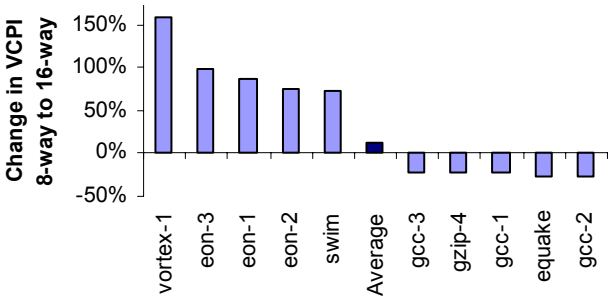

Figure 7. Unpredictability of V_{CPI}.
Benchmarks whose V_{CPI} changes the most in each direction when comparing the 8-way to the 16-way configuration are shown.

If we need to collect additional checkpoints to complete every simulation experiment, we gain no speed advantage from checkpointed warming. Fortunately, this is not the case. First, for an unknown set of experiments, we show that the number of times that we must collect more checkpoints decreases logarithmically with the number of experiments we run, assuring significant long-term time savings from checkpointing. Second, for a known set of experiments, we present a procedure that collects checkpoints at most twice.

Unknown set of experiments. When we perform a series of *m* arbitrary experiments (different microarchitecture configurations) using one benchmark, in the worst case, each experiment exhibits a V_x higher than the previous experiment, and we must collect additional checkpoints for every experiment. However, because V_x varies widely and unpredictably, it is unlikely that we will perform experiments in this worst-case order. If, instead, we assume experiments are performed with a random ordering of V_x, we can use probability theory to determine how often we expect to encounter a new maximum V_x (called a *record*), and thus must collect additional checkpoints. We adapt our argument from the proof in [24], which shows that we expect the number of records to grow logarithmically in *m*. Our probabilistic analysis assumes that we increase our checkpoint library to precisely the size required by the new record on V_x.

The series of *m* randomly-ordered simulation experiments form a random permutation of their V_x: $X = \{V_{x_1}, ..., V_{x_n}\}$. A record (new maximum on V_x) is an element X_j such that $X_j > X_i$ for all $i < j$. The probability that X_1 is a record is one. X_2 is a record if it is the maximum of $\{X_1, X_2\}$. Thus, because the permutation is random, the probability is $1/2$. Generalizing, for X_j, the probability is $1/j$. If we sum the individual probabilities that each

element is a record, we find that the expected number of records over m elements is given by

$$\sum_{j=1}^{m} \frac{1}{j}$$

which is the m^{th} harmonic number. A well-known approximation for harmonic numbers is given by the natural logarithm, ln(m). Hence, the number of times we must collect checkpoints probabilistically grows logarithmically in m.

Known set of experiments. If we wish to run a particular set of m known experiments, we can improve greatly on the ln(m) bound. We can complete all m experiments collecting checkpoints at most twice and with optimal checkpointed simulation time. First, we must collect an initial checkpoint library. The size of this library can be determined based on historical data, or by using SMARTS to run a randomly chosen experiment and obtain a V_x. Then, we run all m checkpointed experiments using the initial library.

Because we chose the initial library size from a random experiment, we expect that, on average, half of the experiments would fail to meet their confidence targets because of insufficient sample sizes. We identify the maximal estimated V_x based on the partial simulation results, and collect additional checkpoints sufficient for this worst-case V_x. This checkpoint set will be large enough for all experiments to achieve the confidence target. We can then resume all partially completed experiments by processing additional checkpoints and combining new results with those of the incomplete runs. The total cost per benchmark for this approach is a single SMARTS run, two checkpoint collections, and m checkpointed simulations of minimal length, independent of m.

5.3 TurboSMARTS procedure

We summarize, here, our complete procedure to experimentation with TurboSMARTS, including steps necessary for the optimizations described in the preceding sections. We separate our procedure for applying TurboSMARTS into three processes: (1) generating checkpoints, (2) managing checkpoint libraries, and (3) estimating performance by processing checkpoints. The first step in generating a checkpoint library is to select an unbiased sample from a benchmark's instruction stream (addressed in Section 5.2). The sample can be a systematic sample as in SMARTS, a simple random sample, or be selected via other sampling methodologies.

The checkpoint generation simulator executes the entire benchmark while continuously updating all warmed structures, as in full warming. When simulation reaches the start of the detailed warming interval for one of the selected measurement locations, we create a new checkpoint. We record snapshots of all structures for which a complete image is required. Then, while simulating the instructions in the warming and measurement intervals, we use a "copy-on-access" policy to capture the correct-path subset of remaining checkpointed structures. As each line or entry of a structure is accessed by a correct-path instruction, we copy the line into the checkpoint. If any system calls occur during the checkpoint interval, we record their effects in the checkpoint. Our checkpoint generation simulator collects checkpoints 20-30% slower than a SMARTS simulation because of the overhead of compressing

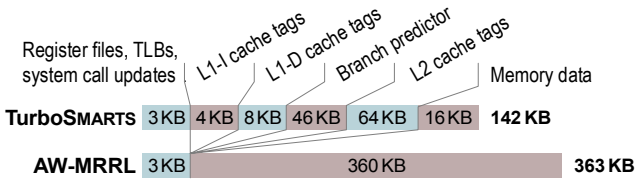

Figure 8. Breakdown of the typical checkpoint, uncompressed. The average full memory footprint was 105 MB for SPEC2K.

checkpoints, resulting in an average checkpoint generation time of 8.5 hours per benchmark.

Once a checkpoint library is generated, we shuffle it into a random order on disk, allowing us to take advantage of the early-stop optimization described in Section 5.1. We shuffle prior to simulation to take advantage of sequential I/O throughput during simulation.

In the event that the checkpoint library is exhausted before the target confidence is reached, we report results with the current confidence and indicate the number of additional required checkpoints. Subsequent simulation can process only the new checkpoints and aggregate new results with those of the initial library, avoiding duplicate checkpoint processing.

6. RESULTS

This section examines the size and processing time breakdown of individual checkpoints in more detail, and discusses how these quantities scale as we increase cache and branch predictor size. We then present performance results of the TurboSMARTS framework and compare it to SMARTS and adaptive warming with MRRL.

6.1 Individual checkpoint performance

The size of TurboSMARTS checkpoint is dominated by the L2 cache tag array and branch predictor state. Architectural state accessed by correct-path instructions comprises only a small fraction of each checkpoint, less than 15% of total checkpoint size. In contrast, architectural state for adaptive warming using MRRL (AW-MRRL) requires 360 KB on average per checkpoint, as all memory values accessed over an average warming period of 4.1 million instructions must be stored. Figure 8 depicts a breakdown of the contents and relative sizes (uncompressed) of a typical checkpoint, assuming a 1MB L2 cache and 1K entry branch predictor.

TurboSMARTS' checkpoint size is highly sensitive to maximum cache and branch predictor size. In contrast, adaptive warming's checkpoints depend solely on application access patterns, and do not grow with larger microarchitecture. Figure 9 (left) shows how typical checkpoint size (uncompressed) scales as we increase the capacity of these structures. `Gzip` compression generally reduces checkpoint size by a factor of 2:1 to 5:1.

As checkpoint size grows, so does the time for loading and decompression. For a 1MB cache, loading and decompression comprise roughly one third of the processing time per checkpoint. For checkpoints over 100 KB (compressed), loading and decompression quickly dominate overall simulation time. This result suggests that optimizing checkpoint I/O (i.e., by using non-blocking I/O operations) might provide considerable speedup for checkpoints

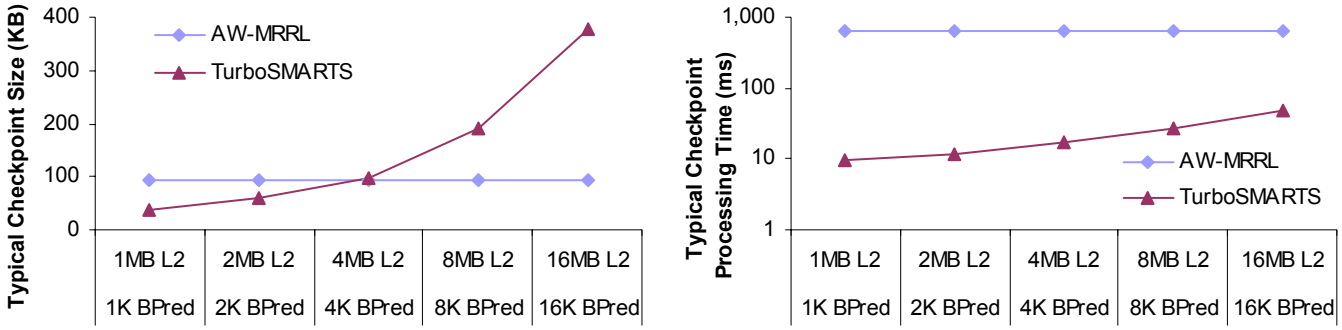

Figure 9. Compressed checkpoint size and processing time. The left and right figures show typical checkpoint size and processing time per checkpoint as the size of checkpointed 8-way processor structures are varied.

that support large caches. Figure 9 (right) shows how checkpoint processing time scales as checkpoint size increases.

6.2 TurboSMARTS performance

We use TurboSMARTS to estimate the CPI of the 41 SPEC2K benchmark/input combinations in our suite with a sample designed to achieve precisely 99.7% confidence of ±3% error in results. Sample size varies from as few as 300 to as many as 60,000 measurements depending on CPI variability in each benchmark. Table 2 presents measured run-time results for our implementation of TurboSMARTS. We present runtimes for non-sampled runs of the complete benchmark with SimpleScalar's `sim-outorder`, full warming using SMARTSim [34], and adaptive warming using MRRL as baselines for comparison. The runtimes assume TurboSMARTS processes all checkpoints serially on a single host CPU. We show the best, average, and worst times for each simulator for the two microarchitecture configurations presented in Section 3.2.

TurboSMARTS reduces average simulation turnaround time from hours for SMARTSim to minutes, with three orders of magnitude speedup over SMARTSim and four orders of magnitude over complete benchmark simulation. TurboSMARTS 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 the benchmark length. Thus, we can expect simulation times to grow with the advent of longer benchmarks (see Figure 2). This is not the case with TurboSMARTS. Instead, runtime depends on sample size, and thus CPI variability. We do not observe any relationship between CPI variability and benchmark length, and thus, we do not expect TurboSMARTS runtimes to increase for longer benchmarks. For example, *sixtrack* is the longest of the SPEC floating-point benchmarks, but can be processed in just seconds because its CPI is stable.

Table 3 presents a summary of the characteristics of each of the warming approaches evaluated in this paper. The table shows the checkpoint library sizes, run times, and non-sampling error measured for each technique. The table also indicates scaling behavior of checkpoint size and runtime with respect to microarchitectural model and benchmark characteristics. Finally, the table indicates what microarchitecture model parameters must be fixed when checkpoints are created, and whether simulations can be parallelized or stopped early.

TurboSMARTS matches the non-sampling error of SMARTS. Adaptive warming with MRRL does not match this tight error. Further increasing the functional warming interval above current adaptive warming may improve accuracy, but will widen the speed gap between adaptive and checkpointed warming. Sampling error can be made arbitrarily small with all three warming approaches by increasing sample size.

TurboSMARTS' checkpoints require fixing maximum cache and TLB sizes and must include state for each branch predictor used in subsequent simulations. However, parameters which do not affect

Table 2. Runtimes of SPEC2K benchmarks.

	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*
SMARTS	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*
TurboSMARTS	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), and seconds (s).

Table 3. Simulation sampling warming methods comparison.

	Sim-outorder	SMARTS	AW-MRRL	TurboSMARTS
Average (worst) CPI non-sampling error	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
*MRRL can be parallelized, but bias increases to 1.9% average, 11% worst.

the organization of these structures are not fixed. Full warming and adaptive warming do not require fixing any simulation parameters.

A SPEC2K checkpoint library for TurboSMARTS can be easily stored on a modern high-capacity disk, and TurboSMARTS simulations are generally not I/O bound when maximum cache size is small. However, TurboSMARTS checkpoints grow with increasing maximum cache size, and will exceed the size of adaptive warming checkpoints for large caches.

TurboSMARTS eliminates the functional warming bottleneck in previously proposed simulation sampling approaches, reducing average simulation time for SPEC2K benchmarks from 7 hours to just 1.5 minutes. TurboSMARTS is approximately 50 times faster than adaptive warming using MRRL to tune the functional warming for each checkpoint.

7. RELATED WORK

Many previous studies of simulation methodology present techniques orthogonal to TurboSMARTS. A variety of programming techniques can accelerate simulators by up to an order of magnitude without affecting simulation results [6,7,19,29,31], however, simulation of complete benchmarks remains expensive. Construction and evaluation of short synthetic benchmarks with statistical properties similar to target workloads, commonly referred to as statistical simulation [25,26], can reduce simulation time to seconds. However, increasing the applicability, robustness and accuracy of these techniques remains an active research topic [8,17]. Some proposals seek to parallelize simulation over many CPUs [10,20,22], improving simulation latency, but provide no benefit to simulator throughput.

TurboSMARTS extends previous work on simulation sampling. Periodic simulation sampling was first proposed in the context of trace-based cache simulation [21]. Conte et al. proposed using sampling theory to explicitly calculate confidence of performance estimates [5]. SMARTS [34] and similar recent work [23] minimize total instructions simulated in detail, and form the basis for the sampling methodology of TurboSMARTS. Other recent sampling proposals employ directed sampling [1,20,32], where the measured sample of a benchmark is chosen based on microarchitecture-independent analysis of program characteristics, such as the relative frequency of static basic blocks. However, directed sampling techniques do not minimize total instructions simulated in detail (unless applied to SMARTS-like sampling unit sizes) while providing quantitative measures of confidence with each result [35].

The TurboSMARTS measurement framework has been successfully integrated into the Liberty Simulation Environment (LSE) by researchers at Princeton University [28]. 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 TurboSMARTS into LSE reduced typical simulation times by up to 20*x* over SMARTS. Moreover, the early stop optimization (see Section 5.1) possible with TurboSMARTS reduced the typical implement-debug-test cycle of model development to less than an hour, greatly accelerating the model development process.

8. CONCLUSION

TurboSMARTS reduces microarchitecture simulation time to the limit imposed by detailed simulation—mere minutes—by applying checkpointing to simulation sampling. TurboSMARTS leverages state-of-the-art simulation sampling techniques to simulate a minimum of instructions in detail by using large sample sizes with very small sampling units of 1000 instructions each. Unlike previous sampling approaches, TurboSMARTS' turnaround time is independent of benchmark length, depending only on the target metric's variance. Therefore, TurboSMARTS is able to simulate benchmarks far longer than those used currently with no increase in simulation time. TurboSMARTS enables complete checkpoint libraries with reasonable storage requirements by storing only necessary functionally-warmed state for several thousand instructions of accurate performance simulation. A reusable checkpoint library for SPEC2K requires only 12 GB. By processing checkpoints in a random order, TurboSMARTS allows simulations to be terminated as soon as a target confidence is reached.

The vast increase in simulation speed provided by TurboSMARTS translates to a much higher experimental throughput. Parametric studies that cover a wide range of microarchitectural options can now be evaluated accurately on entire benchmark suites with reasonable computational requirements. In addition, TurboSMARTS enables interactive performance estimates for individual benchmarks in minutes, enabling quick evaluations of design decisions with immediate performance feedback.

9. REFERENCES

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

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

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[4] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA*, Feb. 2002.

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

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

[7] S. Dwarkadas, J. R. Jump, and J. B. Sinclair. Execution-driven simulation of multiprocessors: Address and timing analysis. *IEEE Transactions on Modeling and Computer Simulation*, Volume 4, No. 4:314–338, Oct. 1994.

[8] L. Eeckhout, R. B. Jr., B. Stougie, K. D. Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the International Symposium on Computer Architecture*, June 2004.

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

[10] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. 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.

[11] J. W. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the International Conference on Computer Design*, Sept. 2001.

[12] J. W. Haskins and K. Skadron. 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.

[13] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

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

[15] International Organization for Standardization. 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 (2002).

[16] R. K. Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 2001.

[17] R. B. Jr., L. Eeckhout, and L. K. John. Debunking statistical simulation in HLS. In *Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking*, June 2004.

[18] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 1991.

[19] V. Krishnan and J. Torrellas. A direct-execution framework for fast and accurate simulation of superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1998.

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

[21] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, Volume C-37(11):1325–1336, Feb. 1988.

[22] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *Hawaii International Conference on System Sciences*, Volume 1: Architecture, pages 205–210, Jan. 1994.

[23] W. Liu and M. Huang. Expert: Expedited simulation exploiting program behavior repetition. In *Proceedings of the International Conference on Supercomputing*, June 2004.

[24] H. M. Mahmoud. *Sorting: A Distribution Theory*. John Wiley & Sons, Inc., 2000.

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

[26] M. Oskin, F. T. Chong, and M. K. Farrens. HLS: combining statistical and symbolic simulation to guide microprocessor designs. In *Proceedings of the International Symposium on Computer Architecture*, June 2000.

[27] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessors performance and simulation methodology. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 72–83, February 1997.

[28] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of flexible validated processor models. Technical Report 04-03, Liberty Research Group, Princeton University, Nov. 2004.

[29] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, May 1993.

[30] M. Rosenblum, S. A. Herrod, E. Witchell, and A. Gupta. Complete computer simulation: The simos approach. *IEEE Parallel and Distributed Technology*, 1995.

[31] E. Schnarr and J. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

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

[33] Standard Performance Evaluation Corporation. SPEC CPU2004 submission requirements. http://www.spec.org/cpu2004/, 2004.

[34] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.

[35] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. An evaluation of stratified sampling of microarchitecture simulations. In *Third Annual Workshop on Duplicating, Deconstructing, and Debunking, ISCA*, June 2004.

[36] J. J. Yi, D. J. Lilja, R. Sendag, S. V. Kodakara, and D. M. Hawkins. Characterizing and comparing prevailing simulation methodologies. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, Feb. 2005.