Two-Level Hybrid Sampled Simulation of Multithreaded Applications

CHUNTAO JIANG, Huazhong University of Science and Technology, China

ZHIBIN YU, Shenzhen Institute of Advanced Technology, China

LIEVEN EECKHOUT, Ghent University, Belgium

HAI JIN and XIAOFEI LIAO, Huazhong University of Science and Technology, China

CHENGZHONG XU, Shenzhen Institute of Advanced Technology, China/Wayne State University, USA

Sampled microarchitectural simulation of single-threaded applications is mature technology for over a decade now. Sampling multithreaded applications, on the other hand, is much more complicated. Not until very recently have researchers proposed solutions for sampled simulation of multithreaded applications. *Time-Based Sampling (TBS)* samples multithreaded application execution based on time—not instructions as is typically done for single-threaded applications—yielding estimates for a multithreaded application's execution time. In this article, we revisit and analyze previously proposed TBS approaches (periodic and cantor fractal based sampling), and we obtain a number of novel and surprising insights, such as (i) accurately estimating *fast-forwarding IPC*, that is, performance in-between sampling units, is more important than accurately estimating *sample IPC*, that is, performance within the sampling units; (ii) fast-forwarding IPC estimation accuracy is determined by both the sampling unit distribution and how to use the sampling units to predict fast-forwarding IPC; and (iii) cantor sampling is more accurate at small sampling unit sizes, whereas periodic is more accurate at large sampling unit sizes.

These insights lead to the development of *Two-level Hybrid Sampling (THS)*, a novel sampling methodology for multithreaded applications that combines periodic sampling's accuracy at large time scales (i.e., uniformly selecting coarse-grain sampling units across the entire program execution) with cantor sampling's accuracy at small time scales (i.e., the ability to accurately predict fast-forwarding IPC in-between small sampling units). The clustered occurrence of small sampling units under cantor sampling also enables shortened warmup and thus enhanced simulation speed. Overall, THS achieves an average absolute execution time prediction error of 4% while yielding an average simulation speedup of 40× compared to detailed simulation, which is both more accurate and faster than the current state-of-the-art. Case studies illustrate THS' ability to accurately predict relative performance differences across the design space.

This work is supported in part by the China National Basic Research Program (973 Program, No. 2015CB352400), National High Tech Research and Development Program of China (863 Program) under Grant No. 2015AA015303, Shenzhen Peacock Innovation Project (KQCX20140521115045448), and National Natural Science Foundation of China under Grants No. U1401258, No. 61322210, No. 61272408, and No. 61433019. Lieven Eeckhout is partly supported by a Chinese Academy of Sciences (CAS) visiting professorship for senior international scientists, and through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013)/ERC grant agreement No. 259295.

David Whalley served as Editor-in-Chief for this submission.

Authors' addresses: C. Jiang, H. Jin, and X. Liao, Services Computing Technology and System Laboratory/ Cluster and Grid Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, No 1037 Luoyu Road, Hongshan District, Wuhan, China; emails: chuntjiang@gmail.com, {hjin, xfliao}@hust.edu.cn; Z. Yu (corresponding author) and C. Xu, Research Center for Cloud Computing, Shenzhen Institute of Advanced Technology, CAS, China; emails: {zb.yu, cz.xu}@siat.ac.cn; L. Eeckhout, Department of Electronics and Information Systems, Ghent University, Belgium; email: lieven.eeckhout@elis.ugent.be.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/11-ART39 \$15.00

DOI: <http://dx.doi.org/10.1145/2818353>

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Hardware** → **Simulation and emulation**; • **Computing methodologies** → *Simulation evaluation*;

Additional Key Words and Phrases: Microarchitecture simulation, sampling, multithreaded workloads, multicore processor

ACM Reference Format:

Chuntao Jiang, Zhibin Yu, Lieven Eeckhout, Hai Jin, Xiaofei Liao, and Chengzhong Xu. 2015. Two-level hybrid sampled simulation of multithreaded applications. *ACM Trans. Archit. Code Optim.* 12, 4, Article 39 (November 2015), 25 pages.

DOI: <http://dx.doi.org/10.1145/2818353>

1. INTRODUCTION

Microarchitectural simulation is an indispensable tool in the computer architect's toolbox, both in academia and industry. Simulation speed has since long been a major challenge. Arguably, the simulation challenge has gotten worse since we entered the multicore era. Most of our simulators are single threaded, yet we need to simulate increasing numbers of cores, which leads to an increasingly widening gap between simulation performance and the performance of the target systems we are modeling in our simulators. Today's simulators are extremely slow, with typical simulation speeds in the hundreds of Kilo Instructions Per Second (KIPS) range for academic simulators [Binkert et al. 2011; Ubal et al. 2012; Patel et al. 2011], and around 1–10KIPS for industry simulators [Emer et al. 2002; Bose and Conte 1998]. Parallelizing architectural simulators to leverage existing multicore processor performance has proved to be a promising acceleration approach [Carlson et al. 2011; Chen et al. 2010; Miller et al. 2010; Reinhardt et al. 1993; Sanchez and Kozyrakis 2013]. However, it is challenging in terms of both software engineering efforts as well as balancing speed versus accuracy.

Sampling is a well-known and widely used simulation acceleration technique that dramatically improves simulation speed by simulating a limited number of *sampling units*, from which overall application performance is estimated. Sampled simulation is mature technology for single-threaded applications on single-core processors [Conte et al. 1996; Sherwood et al. 2002; Wunderlich et al. 2003; Yi et al. 2005], and multiprogram workloads (coexecuting single-threaded applications) on multicore processors [Van Biesbrouck et al. 2004]. These approaches define sampling units by instruction count, that is, a sampling unit refers to a sequence from the dynamic instruction stream as a unit of work, hence the name *Instruction-Based Sampling (IBS)*. IBS employs IPC as the final performance metric of an application, which is extrapolated by only considering the IPC values for the various sampling units.

Sampled simulation of multithreaded applications, on the other hand, is much more challenging, because a unit of work can no longer be defined based on instruction count, that is, the dynamic instruction stream may vary across runs because of nondeterminism and synchronization activities. In other words, IPC is no longer a valid metric; instead, execution time is the ultimate metric for measuring multithreaded workload performance [Alameldeen and Wood 2006]. Researchers have therefore proposed *Time-Based Sampling (TBS)* [Ardestani and Renau 2013; Carlson et al. 2013; Jiang et al. 2013], which estimates a multithreaded application's execution time via sampled simulation. The key idea of TBS is to sample based on time (cycles), not instructions: for example, TBS performs detailed timing simulation for a duration of x microseconds every y microseconds; TBS reverts to fast-forwarding or functional simulation in-between sampling units. Because TBS samples in units of time, a fundamental problem for TBS is that one needs to know how time progresses (in order to know when to sample). Progress is known during the sampling units since detailed simulation is performed—performance observed during the sampling units when a thread is not idling due to synchronization effects is referred to as *sample IPC*. However, progress

is unknown in-between sampling units because TBS performs functional simulation during fast-forwarding, which operates on instructions but does not simulate timing, hence performance is unknown. TBS thus needs to *predict* the performance in-between sampling units. This is done (i) by predicting *fast-forwarding IPC*, that is, the IPC when a thread executes instructions and makes forward progress, and (ii) by determining the impact of idle time as a result of interthread synchronization.

In this article, we first perform a comprehensive evaluation and comparison of existing sampling schemes including periodic, cantor, and random sampling in the context of TBS, which has not been done before and that provides the necessary insight leading to our novel sampling strategy. Periodic sampling [Ardestani and Renau 2013; Carlson et al. 2013] and cantor fractal based sampling [Jiang et al. 2013] have been used in TBS, while random sampling has not. Our evaluation reveals several new and useful insights, some of which may be surprising. We find that (1) all TBS approaches predict sample IPC very accurately; (2) TBS' accuracy is determined primarily by the ability to accurately predict fast-forwarding IPC, not sample IPC; (3) fast-forwarding IPC estimation accuracy is determined by both the sampling unit distribution and how to use the sampling units to predict fast-forwarding IPC; (4) periodic sampling is more accurate at larger sampling unit sizes than random and cantor sampling because it selects large sampling units uniformly across the entire program execution; (5) cantor sampling is substantially more accurate at smaller sampling unit sizes compared to periodic and random sampling because of its clustered sampling units and sophisticated fast-forwarding IPC estimation method; and (6) random sampling is not appropriate for TBS due to variations across different runs.

These novel insights lead to the development of *Two-level Hybrid Sampling (THS)*, a novel TBS approach that significantly outperforms the state-of-the-art. At the first level, THS performs periodic sampling yielding coarse-grain sampling units at fairly large time scale (e.g., 1–10ms). Within the selected coarse-grain sampling units, THS then performs cantor sampling to obtain fine-grain sampling units at a smaller time scale (e.g., 10 μ s). The intuition behind THS is to combine periodic sampling's accuracy at large time scales (i.e., uniformly selecting coarse-grain sampling units across the entire program execution) with cantor sampling's accuracy at small time scales (i.e., ability to accurately predict fast-forwarding IPC in-between small sampling units). An additional, previously unrealized, benefit of cantor sampling is that sampling units are clustered, and hence the fast-forwarding intervals are relatively short compared to periodic sampling. This enables much shorter functional warming of large hardware structures such as caches, TLBs, and branch predictors, prior to sampling units, which leads to a larger overall simulation speed. We propose and explore novel warmup strategies that vary the amount of warmup based on the fast-forwarding interval length.

Based on our experimental evaluation considering an eight-core processor running the PARSEC benchmarks [Bienia et al. 2008] in the Sniper simulator [Carlson et al. 2011], we report an average absolute execution time prediction error of 4% for THS, while yielding an average 40 $\times$ (and up to 58 $\times$) simulation speedup compared to detailed simulation. We compare THS against the state-of-the-art and conclude that THS has both more accurate performance estimation (4.4%, 5.3%, and 6.2% average absolute error for periodic sampling following Carlson et al. [2013], cantor sampling [Jiang et al. 2013], and periodic sampling following Ardestani and Renau [2013], respectively) and higher simulation speed (average speedups of 3 $\times$, 21 $\times$, and 4.5 $\times$, respectively). Applying our novel warmup method to the other three TBS techniques, we find that THS still is the best one in both accuracy and speed. We also evaluate THS' ability to make relative performance predictions and guide practical design decisions: we report that THS is accurate across core count (evaluated up to 32 cores), and microarchitecture changes (core issue width, last-level cache replacement policy, and heterogeneous versus homogeneous multicore configurations).

In particular, we make the following contributions in this article:

- We are the first to perform a comprehensive characterization of existing sampling methods for multithreaded applications including periodic, cantor, and random sampling, and we reveal six novel and useful insights (listed previously).
- We propose THS, a novel technique to accelerate architecture simulation of multithreaded applications by combining periodic sampling at large time scales (first level) and cantor sampling at small time scales (second level).
- We explore and determine appropriate sample sizes as well as adaptive warmup strategies that vary the amount of warmup based on the fast-forwarding length. These optimizations provide a good trade-off between prediction accuracy and simulation speed.
- We give a thorough evaluation for the proposed THS methodology, and we provide a comprehensive comparison of THS against the state-of-the-art.
- We investigate THS' ability to make relative performance predictions and guide practical design decisions.

The remainder of this article is organized as follows. Section 2 describes background and key challenges for TBS. Section 3 analyzes existing TBS approaches, which leads to a number of novel and useful insights regarding important factors affecting TBS accuracy. Section 4 describes THS. After detailing our experimental setup in Section 5, we then present and analyze THS' accuracy and simulation speed in Section 6, followed by a number of case studies illustrating the usefulness of THS in practical design studies (Section 7). Section 8 discusses related work and Section 9 concludes the article.

2. BACKGROUND AND KEY CHALLENGES

Microarchitecture simulation is an indispensable tool in the computer architect's toolbox, yet its slow speed is a major concern, and accelerating simulation remains a grand challenge [Yi et al. 2006]. Sampling is a well-known and widely used simulation acceleration technique. The key idea is to determine a number of *sampling units* and only simulate those in detail, from which performance is extrapolated to the entire execution. We identify three modes of simulation during sampled simulation. During *detailed simulation* of the sampling units, we simulate the entire processor in detail, that is, timing simulation of all hardware structures. Navigating in-between sampling units is done via *fast-forwarding*, that is, functional simulation to transfer to the correct architecture state at the beginning of the next sampling unit. Because the microarchitecture state is unknown at the beginning of a sampling unit, *functional warming* may be used to "warm up" large hardware structures, such as caches, TLBs, branch predictors, etc. Functional warming does not collect performance results; its sole purpose is to make sure the hardware state at the beginning of a sampling unit is an accurate representation of the hardware state as if we would have simulated the entire processor in detail up until the sampling unit.

Detailed simulation is the slowest among the three simulation modes, followed by functional warming and fast-forwarding. The simulation speed difference between detailed simulation and functional warming is a factor $10\times$ in our simulation setup; the simulation speed difference between functional warming and fast-forwarding is a factor $100\times$. Hence, to achieve high simulation speed, it is important to limit the number of sampling units and their size; likewise, functional warming should be minimized so that high simulation speeds are obtained while not compromising accuracy too much.

2.1. IBS versus TBS

As mentioned before, IBS selects sampling units guided by instruction count, and is a widely used approach for single-threaded workloads. It only considers sampling units

to predict the average IPC as the final performance metric of an application. This approach is built on the notion that the dynamic instruction stream of a single-threaded application does not change across different runs, making IPC a valid performance metric for evaluating single-core processor performance. However, as aforementioned, due to nondeterminism and synchronization effects, the dynamic instruction stream of a multithreaded application varies from run to run, making IPC an inappropriate metric [Alameldeen and Wood 2006], and thus IBS is considered harmful for multithreaded applications.

TBS [Ardestani and Renau 2013; Carlson et al. 2013; Jiang et al. 2013], on the other hand, selects sampling units based on execution time, rather than instruction count, to estimate a program's overall execution time. This approach is motivated by the observation that not IPC, but execution time is the ultimate metric for measuring multithreaded application performance [Alameldeen and Wood 2006]. A fundamental problem with TBS thus is to estimate performance for both the sampling units and the nonsampling units, in order to be able to predict an application's overall execution time. Estimating performance during a sampling unit is straightforward since we are simulating the processor in detail. Estimating performance in-between sampling units, on the other hand, is nontrivial.

There are two key issues related to estimating performance in-between sampling units. The first issue is to predict performance when a thread is making forward progress (nonidle), that is, a thread is executing useful instructions. TBS does so by predicting a thread's *fast-forwarding IPC* based on the nonidle IPC of (a) prior sampling unit(s). The second issue relates to how interthread synchronization effects affect performance. TBS solves this issue by keeping track of synchronization effects, by intercepting mutex and futex events (e.g., `pthread_mutex`, `futex` system calls, etc.), and by accurately modeling how these synchronization events affect threads' progress during fast-forwarding. For example, when a thread is waiting for a lock and turns idle, it no longer executes instructions; later, when the lock is released, the stalled thread awakes and inherits the time of the thread that released the lock. This enables TBS to accurately estimate the effect of synchronization in-between sampling units. Once all threads have advanced time till the end time of the in-between sampling unit, only then does the simulator switch to the next simulation mode.

2.2. Key Challenges for TBS

TBS introduces a number of challenges compared to IBS. First of all, IPC overestimates and underestimations per sampling unit in IBS do not always increase the overall IPC prediction error because they might cancel each other out when calculating an average IPC of the whole application. In TBS, on the other hand, an underestimation or overestimation of the performance of both sampling units and in-between sampling units leads to a thread underestimating or overestimating its performance, which in its turn affects what is the next unit of work that is to be simulated (because sampling is time based). In other words, performance estimation errors may accumulate over time, yielding nonrepresentative sampled simulation runs. Moreover, even relatively small timing differences may lead to different thread interleavings, lock acquisition, distorted timing at barriers, etc. Some of the relative timing differences may not affect simulation accuracy at all; for example, underestimating the IPC of a noncritical thread ahead of a barrier does not affect when the program reaches the barrier, which is determined by the lagging, most critical thread. In other cases, mispredictions for the critical thread may lead to overall program performance prediction inaccuracies.

Another major challenge is a result of the difference in size between a sampling unit versus the in-between sampling units. Fast-forwarding IPC is predicted based on the previous sampling unit's IPC. In other words, a relatively small interval's IPC

Fig. 1. Illustrations for different sampling strategies: cantor, periodic, and (simplified) random sampling.

(sampling unit) is used as a prediction for a much larger interval's IPC (fast-forwarding or in-between sampling unit). The problem now is that there is more performance variability at small time scales compared to bigger time scales [Duesterwald et al. 2003; Sherwood et al. 2001], which makes achieving accurate predictions for fast-forwarding IPC challenging. Yet, TBS requires accurate fast-forwarding IPC predictions as we argued before, and as we will quantitatively study in this article.

3. ANALYZING EXISTING SAMPLING TECHNIQUES

As mentioned previously, compared to IBS, TBS is still immature technology, leaving several issues unresolved. Our goal in this section is to provide a comprehensive analysis and comparison of previously proposed sampling schemes in the context of TBS. Several useful, and perhaps surprising, insights will be revealed, which is helpful to better understand TBS, and eventually leads to the novel THS approach outlined in the next section.

3.1. TBS Techniques

Before doing so, we first briefly revisit these sampling methods. Periodic sampling [Ardestani and Renau 2013; Carlson et al. 2013] and cantor sampling [Jiang et al. 2013] have been proposed before. Random sampling, to the best of our knowledge, has not yet been discussed before in the context of TBS. However, we still consider it here for completeness.

3.1.1. Cantor Sampling. Cantor sampling [Jiang et al. 2013] selects sampling units according to the generation rule of a Cantor set—a classical fractal [Peitgen et al. 2004]. As illustrated in Figure 1(a), by treating the execution time as a line segment, cantor sampling recursively selects the first third and the last third of the line segment as candidates. This process is iterated for a number of so-called *division steps* ($K = 4$ in the example). The final sampling units are shown as solid black bars in Figure 1(a) and are simulated in detail; the time intervals between the sampling units are simulated in fast-forwarding mode and/or (partially) via functional warming (i.e., warming structures such as caches, TLBs, branch predictors, etc.). Stopping after K division steps leads to 2^K sampling units in total. The *Fraction of Detailed Simulation (FDS)*, that is, the

ratio of what needs to be simulated in detail versus total time, equals:

$$FDS_{cantor} = \left(\frac{2}{3}\right)^K. \quad (1)$$

Fast-forwarding IPC is computed by considering the sampling unit(s) that fall(s) within the prior time interval of the same length as the fast-forwarding interval. For the example in Figure 1(a), the fast-forwarding IPC of interval #2 is computed as the sample IPC of interval #1; the fast-forwarding IPC of interval #4 is computed as the average sample IPC of sampling units #1 and #3; the fast-forwarding IPC of interval #6 is estimated as the sample IPC of interval #5; for interval #8, we estimate the fast-forwarding IPC as the average IPC of sampling units #1, #3, #5 and #7, etc. This can be generalized by the following equation: Fast-forwarding IPC is estimated by averaging the IPC of the q latest sampling units, with q computed as

$$q = 2^{\lceil \log_3(\frac{L_i}{D}) \rceil}, \quad (2)$$

with D the sampling unit size, and L_i the length of the i^{th} fast-forwarding interval (in units of D). For the example in Figure 1(a), considering fast-forwarding interval #16, L_i/D equals 27, yielding $q = 8$, which means we need to consider the eight last sampling units to compute #16's fast-forwarding IPC. We compute fast-forwarding IPC as the (arithmetic) mean of the sampling units' IPC [John 2006].¹

3.1.2. Periodic Sampling. Periodic sampling [Ardestani and Renau 2013; Carlson et al. 2013], as the name suggests, periodically selects sampling units to be simulated in detail (see Figure 1(b)). It involves two parameters: D and F , with D the sampling unit size, and F the fast-forwarding interval size in units of D . For example, " $D = 10\mu s$, $F = 4$ " means that a sampling unit size of $10\mu s$, and a fast-forwarding interval length of " $4 \times 10\mu s = 40\mu s$ ". The FDS for periodic sampling equals

$$FDS_{periodic} = \frac{1}{F + 1}. \quad (3)$$

Fast-forwarding IPC of a thread is estimated by the sample IPC of the last sampling unit, as suggested by previous work [Ardestani and Renau 2013; Carlson et al. 2013]. For example, the fast-forwarding IPC of interval #2 is estimated as the sample IPC of interval #1; fast-forwarding IPC of interval #4 is estimated as the sample IPC of sampling unit #3, etc.

3.1.3. Random Sampling. We consider two random sampling methods, which are adapted from the random sampling approach proposed by Conte et al. [1996]: S-random and m-random. In S-random, we first divide total execution time into fairly large equal-length intervals. Each such interval is further divided into P smaller intervals with equal length. We then randomly select one from these P small intervals to simulate in detail. In the example given in Figure 1(c), we assume $P = 5$, or in other words, we randomly select one out of five intervals, every five intervals. The FDS for S-random sampling is calculated as follows:

$$FDS_{S-random} = \frac{1}{P}. \quad (4)$$

Fast-forwarding IPC is computed similarly to periodic sampling by assuming it equals the sample IPC of the last sampling unit, for example, interval #2's fast-forwarding IPC is estimated as interval #1's sample IPC.

¹The arithmetic mean is the correct mean because time is constant across all sampling units.

Table I. Parameters for the Different Sampling Methods During the Trace-Driven Experiments

	Sample size ($D = 10\mu s$)	Fast-forwarding parameter (K , F , or P)								
Cantor	1,2,3,4,5,6,7,8,9,	K	3	4	5	6	7	8	9	10
Periodic	10,20,30,40,50,60,70,80,90,	F	2	4	6	10	16	24	37	56
S-random	100,200,300,400,500,600,700,800,900.	P	3	5	7	11	17	25	38	57

S-random has controlled parameters so that we are able to provide a fair comparison with cantor and periodic sampling. However, it may be considered not “random enough.” We therefore also consider m-random sampling, which makes the sampling units start time at any random time for a random duration. In addition, we use the average IPC of a random number of latest sampling units to predict the fast-forwarding IPC.

3.2. Trace-Driven Analysis

Ideally, comparing these previously proposed sampling methods should be done via execution-driven experiments. We find this to be infeasible though. The design space of these sampling methods is huge as there are several parameters that need to be set and explored for each of these methods; in particular, we vary the cross section of sampling unit size and fast-forwarding interval size, which leads to a couple hundred parameter settings. Evaluating all of these possible sampling configurations via execution-driven simulation would be simply too time consuming. In addition, and more importantly, execution-driven experiments would not enable us to answer the important question of whether a particular sampling strategy is particularly effective at predicting sample IPC versus fast-forwarding IPC.

For these reasons, we revert to trace-driven evaluation, for now. Later, in Section 6, we will be evaluating and comparing THS against other sampling methods via execution-driven simulation. To enable a fair comparison, we implement all three sampling methods in the same simulation environment using the same workloads. We collected traces by running the PARSEC benchmarks [Bienia et al. 2008] with eight threads on an eight-core system using Sniper [Carlson et al. 2011]. (See Section 5 for a more detailed description of our experimental setup and the modeled multicore processor architecture.) For each of these benchmarks, we store (nonidle) IPC results on disk per $10\mu s$, for each thread. This enables us to quickly assess a sampling method’s accuracy as we vary sampling parameters. The sampling parameters for each sampling method in our experiments are shown in Table I. (We consider S-random here, and evaluate m-random in Section 3.6.) Sample size D is counted in units of $10\mu s$, and varies over a broad range from $10\mu s$ to 9ms. The fast-forwarding parameters K , F , and P are different for the three methods, and are chosen such that they lead to approximately the same FDS, enabling a fair comparison. For example, setting the parameters $(K, F, P) = (7, 16, 17)$ for cantor, periodic, and S-random sampling, respectively, leads to an FDS of (approximately) 5.8%.

3.3. Estimating Sample IPC versus Fast-Forwarding IPC

As mentioned before, the golden rule to measure a TBS method is how accurately it can estimate one program’s execution time. TBS estimates the overall execution time using both sample and fast-forwarding time (cycles). Hence, it is important to accurately estimate sample IPC and fast-forwarding IPC, which can help to know how time progresses, then determining the overall estimation accuracy. We now evaluate how effective the various TBS methods are in this regard. We set $D = 10\mu s$, and we assume $(K, F, P) = (7, 16, 17)$ for cantor, periodic, and S-random sampling, respectively.

Fig. 2. Sample IPC error (left graph) and fast-forwarding IPC error (right graph) for cantor, periodic, and S-random sampling, assuming $D = 10\mu s$ and $(K, F, P) = (7, 16, 17)$.

We quantify the *sample IPC error* as the absolute error between the IPC computed over the sampling units versus the overall IPC:

$$IPC_{sample\ error} = \left| \frac{\sum_{i=1}^n INS_i}{\sum_{i=1}^n CYC_i} - IPC_{overall} \right|, \quad (5)$$

with INS_i the number of instructions in the i^{th} sampling unit, CYC_i the number of cycles in the i^{th} sampling unit, n the number of sampling units, and $IPC_{overall}$ the overall IPC for the entire benchmark execution (obtained by simulating the entire benchmark via detailed simulation). In other words, the sample IPC error is computed as the absolute error of the average per-sample IPC versus the overall IPC. The sample IPC error is thus a measure for how well the sampling units are able to predict overall application performance *under the assumption that the fast-forwarding IPC is estimated with perfect accuracy*.

As noted before, it is critical for a TBS method to also accurately estimate fast-forwarding IPC, for which we define the *fast-forwarding IPC error*:

$$IPC_{ffwd\ error} = \sum_{i=1}^n \left(\left| IPC(F)_i - \widetilde{IPC(F)}_i \right| \times \frac{L(F)_i}{L_{overall}} \right), \quad (6)$$

with $IPC(F)_i$ the IPC of the i^{th} fast-forwarding interval, $\widetilde{IPC(F)}_i$ the estimated IPC of the i^{th} fast-forwarding interval, $L(F)_i$ its length (counted in microseconds), and $L_{overall}$ the total benchmark's execution time (length). The estimated IPC for each fast-forwarding interval is computed as described in Section 3.1. The fast-forwarding IPC error weights the absolute error per fast-forwarding interval with its relative length, and thus quantifies the average absolute error for estimating fast-forwarding IPC.

This analysis leads to three interesting findings.

Finding#1: *All three TBS methods (very) accurately estimate sample IPC.* Figure 2(a) quantifies sample IPC error per benchmark for the three TBS methods. The overall conclusion is that all three TBS methods very accurately predict sample IPC, that is, the error is consistently below 1.3%, with an average absolute error below 0.4% for all three methods. Cantor sampling is less accurate compared to the other sampling techniques for the majority of the benchmarks, and on average. The reason is that cantor sampling, in contrast to periodic and random sampling, does not select sampling units uniformly across the entire benchmark execution. Sampling units tend to be clustered, which leads to some program phases to be “overrepresented” in the average IPC score, and others “underrepresented.” Hence, its average IPC estimate is less accurate.

Finding #2: *TBS inaccuracies primarily result from inaccurate fast-forwarding IPC estimates, not sample IPC inaccuracies.* Figure 2(b) quantifies the fast-forwarding IPC error. This result leads to a novel interesting, and perhaps surprising, insight. The

Fig. 3. Normalized fast-forwarding IPC error (normalized to the largest error observed for a given fast-forwarding length) as a function of sample size D (measured in units of $10\mu s$) for cantor, periodic, and S-random sampling.

fast-forwarding IPC error is much higher than the sample IPC error, that is, the average fast-forwarding IPC error varies between 5% and 10% with outliers up to 43%. (Compare this to less than 1.3% errors for the sample IPC.) In other words, the inaccuracies introduced by all three TBS methods do not come from inaccurately estimating sample IPC, but from inaccurately estimating fast-forwarding IPC. Therefore, being able to accurately estimate fast-forwarding IPC is more important than being able to accurately estimate sample IPC. While this may have been rather surprising at first, it is quite intuitive in hindsight. The largest fraction of the total benchmark execution is simulated in fast-forwarding mode under TBS, and overall multithreaded application performance is estimated using predictions of both the sampling units *and* the fast-forwarding intervals.

Finding #3: Fast-forwarding IPC estimation accuracy is determined by both the sampling unit distribution and the fast-forwarding IPC computation method. It is interesting to find that, although cantor sampling has higher sample IPC error (Figure 2(a)), it is able to provide substantially more accurate fast-forwarding IPC estimation compared to periodic and random sampling (see Figure 2(b), around 5% versus 10%). The sampling units are clustered under cantor sampling; they are not uniformly distributed as for periodic and random sampling. Because of that, cantor sampling features a sophisticated fast-forwarding IPC estimation method to take advantage of its unique sampling unit distribution, which leads to higher fast-forwarding IPC estimation accuracy. We therefore conclude that fast-forwarding IPC estimation accuracy is determined by both the sampling unit distribution and the fast-forwarding IPC computation method. See Section 3.5 for an extended analysis and discussion.

3.4. Sensitivity to Sample Size

Next to understanding TBS' sensitivity to accurately estimating sample IPC and fast-forwarding IPC, we now study TBS' sensitivity to sample size. To obtain general conclusions, we consider all possible fast-forwarding interval lengths per sample size, that is, we consider all possible values for (K, F, P) for each value of D (see also Table I), across which we compute an average error metric. The error metric considered in this experiment is the fast-forwarding IPC error as defined previously; the reason for considering this error metric is because it correlates highly with a sampling method's overall accuracy, as concluded from the previous section. We report normalized error metrics to the maximum error observed across all fast-forwarding interval lengths and sampling methods. Figure 3 shows how fast-forwarding IPC error is affected by sample

Fig. 4. Fast-forwarding IPC computation method exploration. Normalized fast-forwarding IPC error (normalized to the largest error observed for a given fast-forwarding length) as a function of sample size D (measured in units of $10\mu s$) for cantor sampling versus periodic-last- X sampling (X means using the prior X sampling unit(s) to predict fast-forwarding IPC).

size. Accuracy (almost) monotonically increases for periodic and random sampling as sample size increases. This is to be understood intuitively as performance variability decreases with larger interval sizes. Recall that for periodic and random sampling, a sampling unit's performance is used to predict the performance of the next fast-forwarding interval, hence a larger sampling unit's IPC is a more accurate predictor for the next fast-forwarding interval.

Finding #4: Periodic sampling is more accurate at larger sampling unit sizes than cantor and random sampling. The reason is that the sampling units are more uniformly distributed across the entire program execution. There are relatively few sampling units, and hence it is important that the sampling units are chosen such that they are representative for the entire program execution. This is also the reason why cantor sampling is the worse of all sampling techniques for large sampling unit sizes.

Finding #5: Cantor sampling is substantially more accurate at smaller sampling unit sizes than periodic and random sampling. Cantor sampling is by far the most accurate sampling technique at small sampling unit sizes (smaller of a couple hundreds of μs). This is due to how it selects sampling units, and how it estimates fast-forwarding IPC, which is the subject of the next section.

3.5. Sensitivity to Sampling Unit Distribution and Fast-Forwarding IPC Estimation Method

Cantor sampling is the only TBS technique that uses multiple previous sampling units to estimate fast-forwarding IPC. In this section, we apply such method to periodic sampling as an example to better understand how the sampling unit distribution and the fast-forwarding IPC estimation method affect accuracy.

Figure 4 shows the fast-forwarding IPC error curves as a function of sample size for cantor sampling and periodic sampling with different fast-forwarding IPC estimation methods. *Periodic-Last- X* (X is chosen as 1, 2, 4, 6, 10, 15, or 20) means that using the average (nonidle) IPC of the last X sampling units as the next fast-forwarding IPC for periodic sampling. The experimental results reveal a couple of interesting observations. For one, considering multiple sampling units for predicting fast-forwarding IPC improves accuracy somewhat for periodic sampling using small sampling unit sizes (up to a couple hundreds of μs); yet, it cannot achieve the same level of accuracy as cantor sampling. The reason is that cantor sampling estimates IPC of a given fast-forwarding interval by computing the average IPC over sampling units selected within the

Fig. 5. Fast-forwarding IPC error bars and sample IPC error bars for S-random (left graph) and m-random (right graph) sampling across 20 times runs with the same parameter configurations. The average error is shown along with the minimum and maximum error for each benchmark.

previous equal-length time interval (which it is able to do because of how its sampling units are clustered). This sophisticated method leads to more robust estimation results at small time scales, decreasing the risk of using too many sampling units to estimate the IPC of a short length of fast-forwarding interval, or vice versa. However, for periodic sampling, it is difficult to determine how many previous sampling units should be used to achieve the best IPC estimation. For example, using too many sampling units (*Periodic-Last-15* or *Periodic-Last-20* in Figure 4) does not provide any benefit but instead even obtaining worse estimation accuracy (compared to *Periodic-Last-6*). Additionally, there is no way to deploy cantor’s IPC estimation method to periodic (or S-random) sampling because of their different sampling unit distributions.

It is also interesting to note that using multiple sampling units to predict fast-forwarding IPC leads to decreased accuracy for periodic sampling with large sampling unit sizes. Program execution behavior is relatively stable at coarse time scales, hence the last sampling unit’s IPC is a good indicator for the IPC of the upcoming fast-forwarding interval. In other words, for large sampling unit sizes, considering the last sampling unit to predict fast-forwarding IPC is the best choice for periodic sampling.

3.6. Variations Across Different Runs

For a comprehensive analysis, we further study the variations across runs for the same workload and configuration. This is important because it determines if a sampling method is reliable or not. In particular, if the estimated results of a sampling method vary in a large range from run to run, architects cannot tell whether the divergency among the two runs is due to different processor configurations or due to the sampling method itself, which is unacceptable.

Cantor and periodic sampling select almost the same time intervals as its sampling units across different runs, which leads to very small estimation accuracy variations across different runs. We therefore focus on random sampling in this section. For S-random sampling, we set $D = 40\mu s$, $P = 17$ and we use the last five sampling units to estimate the IPC of the next fast-forwarding interval. For m-random sampling, it has the same number of sampling units with S-random; we randomly select each sampling unit size in the range of $10\text{--}70\mu s$, and we select each sampling unit’s starting time at random as well; moreover, for each fast-forwarding interval, using the last Y sampling unit(s) to perform its IPC estimation (Y is also selected from the range of $1\text{--}9$ randomly each time).

Finding #6: Random sampling is inappropriate for TBS due to variations across different runs. Figure 5 plots fast-forwarding IPC error and sample IPC error—average, maximum, and minimum—for S-random and m-random sampling. The experimental results are collected by running S-random and m-random sampling 20 times under

Fig. 6. THS overview: We perform periodic sampling at the first level, followed by cantor sampling at the second level.

the exact same sampling parameters. As can be seen, the absolute values and variations of sample IPC error are very small for both random methods (less than 0.5%). However, the fast-forwarding IPC error is much higher and varies in a large range (the average variations from different runs for S-random and m-random could be up to 25.3% and 30.5%, respectively). This is harmful because architects cannot obtain reliable evaluations using such a sampling method. Of course, one may be able to get reasonable results by considering more than 30 runs according to the central limit theorem. However, this is too time consuming to be acceptable practice in microarchitecture simulation. Therefore, we conclude that random selection of sampling units is inappropriate for TBS.

4. TWO-LEVEL HYBRID SAMPLING

We are now able to introduce THS. The key idea behind THS is to deploy a two-level sampling approach: we apply periodic sampling at a coarse time scale granularity (in the millisecond range) and cantor sampling at a much smaller time scale granularity (tens of microseconds), based on the insights described in the previous section. Figure 6 provides an overview of the sampling methodology. Assuming the entire program execution time takes a total time (length) of L_{total} , we first perform periodic sampling, that is, we select a total of N coarse-grain sampling units uniformly distributed across the entire program execution. More specifically, we consider N periodic intervals of length L_p (i.e., $L_{total} = N \times L_p$); each periodic interval is further split up in a coarse-grain sampling unit of length L_{pd} and a fast-forwarding unit of length L_{pf} . As for periodic sampling (see Section 3.1.2), the fast-forwarding factor F defines the relationship between the fast-forwarding interval and the detailed interval, that is, $L_{pf} = F \times L_{pd}$. A fraction of the fast-forwarding interval of length L_{pw} is considered for warming up microarchitecture structures prior to the detailed sampling unit, as we will describe in more detail in Section 4.2.

The second stage then applies cantor sampling within the selected coarse-grain sampling units of length $L_c = L_{pd}$. After K division steps, we end up with 2^K fine-grain sampling units of length L_{cd} . The fast-forwarding intervals between the fine-grain sampling units have variable length L_{cf} per the cantor sampling division procedure. As for the coarse-grain sampling unit, a fraction of the fast-forwarding interval of length

L_{cw} is considered for warming up microarchitecture structures prior to each fine-grain sampling unit, as we will describe in more detail in Section 4.2.

Fast-forwarding between sampling units is done following the procedures for cantor and periodic sampling, respectively. At the second stage, we follow cantor sampling's approach to computing fast-forwarding IPC (see Section 3.1.1). At the first stage, we employ periodic sampling's mechanism to compute fast-forwarding IPC, that is, we consider the average IPC of the past coarse-grain interval as an estimate for the next fast-forwarding interval.

We now describe how the THS parameters relate to each other. The subsequent section then describes warm-up. Finally, we describe how we determine the THS parameters in practice.

4.1. THS Parameters

The fine-grain sampling unit's length (at the second stage) is L_{cd} . For a given value of K for the cantor sampling stage, the coarse-grain sampling unit's length (at the first stage) is then computed as

$$L_c = L_{pd} = L_{cd} \times 3^K. \quad (7)$$

Assuming a fast-forwarding factor F and number of samples N for the periodic sampling stage, total execution time (length) equals

$$L_{total} = N \times (F + 1) \times L_{pd}. \quad (8)$$

The THS parameters thus are L_{cd} , K , F , and N ; variable L_{total} is a dependent variable. As for L_{cd} , we explore THS' accuracy as a function of this parameter in Section 6.1, and we eventually set it to $L_{cd} = D = 10\mu s$ in our experimental framework. We set $F = 8$ following Carlson et al. [2013] regarding coarse-grain periodic sampling. The two remaining parameters, K and N , are related: for a given L_{cd} and F , a higher K implies a smaller N , and vice versa. We evaluated THS' accuracy as a function of N (and thus K), and found THS to yield stable results for N in the 10–1,000 range. In our experimental framework, we thereby choose K such that N is around 10 to achieve less FDS and higher simulation speedup.

Using the preceding formulas, we can easily derive THS' FDS:

$$FDS_{THS} = \frac{1}{F + 1} \times \left(\frac{2}{3}\right)^K. \quad (9)$$

4.2. Warmup

A well-known challenge for sampled simulation is that hardware state is unknown at the beginning of each sampling unit—the *cold-start problem*. The largest hardware structures, such as caches and (to a lesser degree) TLBs and branch predictors, are most susceptible to the cold-start problem. Clearly, the highest accuracy is achieved by keeping all hardware structures warm between sampling units (i.e., functional warming, or accessing caches, TLBs and predictors), however, this has a detrimental effect on simulation speed. We therefore explore warmup strategies that enable speeding up sampled simulation without sacrificing accuracy too much.

An important feature of cantor sampling is that sampling units are clustered, which adds an interesting (unexpected, and previously unexplored) benefit for warmup. Because of the clustering of fine-grain sampling units, the fast-forwarding intervals in-between sampling units are relatively small, which leads to short warmup intervals and thus high simulation speed. Table II summarizes the warmup strategies considered in this article: W_0 , W_1 , W_3 , W_9 , W_{27} , W_{81} , W_{243} , and W_{full} (the numbers here are all in units of sample size D). Each of these warmup strategies consider

Table II. L_{cw} and L_{pw} Values for Different Warmup Strategies W_X
(the Numbers of X , L_{cf} , L_{pf} , L_{cw} , and L_{pw} are All in Units of Sample Size D)

Fast-forwarding interval	L_{cf}						L_{pf}
	1	3	9	27	81	243	729×8
Warmup strategy	L_{cw}						L_{pw}
W-full	1	3	9	27	81	243	729×8
W-243	1	3	9	27	81	243	243
W-81	1	3	9	27	81	81	81
W-27	1	3	9	27	27	27	27
W-9	1	3	9	9	9	9	9
W-3	1	3	3	3	3	3	3
W-1	1	1	1	1	1	1	1
W-0	0	0	0	0	0	0	0

Table III. The Values of $\tilde{K}$ Corresponding to the Ranges of $\widetilde{L}_{total}$

$\tilde{K}$	4	5	6	7	8	9
$\widetilde{L}_{total}$ (s)	0.04–0.13	0.13–0.38	0.38–1.14	1.14–3.41	3.41–10.23	10.23–30.68

different warmup lengths for different fast-forwarding lengths. Full warmup (W_full) assumes that warming is enabled during the entire intersampling unit interval. The other warmup strategies (W_243 through W_0) consider increasing shorter warmup lengths. W_X means that the largest warmup interval length is never larger than X , even if the fast-forwarding interval is longer. We will evaluate these warmup strategies in Section 6.2, and study the trade-off in simulation speed versus accuracy.

4.3. THS in Practice

Having set some of the THS parameters (i.e., $L_{cd} = D = 10\mu s$, $F = 8$, $N = 10$), as mentioned in Section 4.1, we are left determining parameter K . Unfortunately, K is a function of the application's total execution time L_{total} , which is unknown, and which is what we need sampled simulation for in the first place.

To break this cyclic dependence, we employ the following approach. We first estimate $\widetilde{L}_{total}$ via ultrafast simulation by assuming that all threads make progress at a one-IPC rate. This mode of simulation is 200 times faster than detailed simulation and needs to be done only once. Having obtained an estimate $\widetilde{L}_{total}$, and assuming $N = 10$ and $F = 8$, we can compute an estimate for the coarse-grain sampling unit size using a top-down approach:

$$\tilde{L}_c = \tilde{L}_{pd} = \frac{\widetilde{L}_{total}}{N \times (F + 1)}. \quad (10)$$

The next step is to estimate $\tilde{K}$, assuming $L_{cd} = D = 10\mu s$:

$$\tilde{K} = \log_3 \left(\frac{\tilde{L}_c}{L_{cd}} \right). \quad (11)$$

Since $\tilde{K}$ is a noninteger value, we approximate it to the closest integer value K . Note that, the estimation error between $\widetilde{L}_{total}$ (via one-IPC simulation) and L_{total} (via detailed simulation) is 32.6% on average (106.7% max), which is accurate enough to get a reasonable parameter $\tilde{K}$. Table III shows different values of $\tilde{K}$ corresponding to different ranges of $\widetilde{L}_{total}$. The variations of $\widetilde{L}_{total}$ is up to 200% for the same $\tilde{K}$. Therefore,

Table IV. Simulated System Characteristics

Component	Parameters
Processor	Four cores per socket
Core	2.66GHz, four-way issue, 128-entry ROB
Branch predictor	Pentium M [Uzelac and Milenkovic 2009], eight cycles penalty
L1-I cache	32KB, four-way, four cycles access time
L1-D cache	32KB, four-way, four cycles access time
L2 cache	256KB, eight-way, eight cycles
L3 cache	8MB, 16-way, 30 cycles
Main memory	65ns access time, 8GB/s per socket

in the vast majority of cases, using $\widetilde{L}_{total}$ and L_{total} will lead to the exact same value of K . Even the estimation causes a neighboring $\widetilde{K}$ (compared with K obtained by L_{total}) in a few cases; it will not affect the sampling accuracy according to our observation.

Once we obtain this K value, we can then determine the sampled simulation strategy in a bottom-up approach: knowing the fine-grain sampling unit size ($D = 10\mu s$) and K , this determines the coarse-grain sampling unit size L_c (see the formulas in Section 4.1); assuming $F = 8$, this determines the interval length L_p at the first THS level.

5. EXPERIMENTAL SETUP

We implement THS in the parallel, execution-driven multicore simulator Sniper [Carlson et al. 2011], version 6.0. We configure it to model a multisocket, multicore processor with four out-of-order cores per socket. Each core is four-wide superscalar running at 2.66GHz. The cache hierarchy has three levels: the L1 and L2 cache are private per core, and the last-level L3 cache is shared by all four cores on the socket. In the evaluation experiments of THS' relative accuracy, we simulate one-, two-, four-, and eight-socket shared-memory machines, which corresponds to 4-, 8-, 16-, and 32-core systems. See Table IV for the main characteristics of the simulated systems.

The 12 benchmarks used in this work are taken from the PARSEC 2.1 benchmark suite [Bienia et al. 2008] with the simlarge input data size. We failed to run x264 because of limitations within Sniper. In our measurements, only the parallel Region of Interest (ROI) of each benchmark is included. The fast-forwarding simulation mode is used to skip over the initialization and cleanup phases. Sampled simulation of single-threaded workloads has been studied extensively over the past two decades (see, e.g., Conte et al. [1996], Sherwood et al. [2002], Wunderlich et al. [2003], and Yu et al. [2010]), which is why we do not consider the sequential phases here.

6. RESULTS AND ANALYSIS

We now evaluate THS' accuracy and simulation speed. This is done in a number of steps. We first identify the appropriate sample size, after which we explore the impact of the warmup strategy on accuracy and simulation speed. Once we have determined sample size and warmup, we then evaluate THS' overall accuracy and simulation speed; we evaluate both absolute and relative accuracy. Finally, we present a number of case studies illustrating that THS is indeed useful and accurate enough for making real-life design decisions.

6.1. Sample Size

As we discussed extensively before, sample size has a significant impact on sampling accuracy. We therefore perform an exploration to understand THS' sensitivity to sample size. We assume full functional warmup, that is, caches, TLBs, and branch predictor are

Fig. 7. Average absolute prediction error for THS as a function of sample size. Full functional warmup is assumed between samples.

Fig. 8. Warmup strategies for THS: simulation speedup over detailed simulation (vertical axis) versus accuracy (average absolute execution time prediction error on the horizontal axis).

kept warm (updated) between subsequent samples, and we only vary sample size while keeping the other sampling parameters fixed (i.e., $N = 10$ and $F = 8$). We consider nonideal warmup in the next section.

Figure 7 plots the average absolute error (for predicting execution time) for the 12 PARSEC benchmarks as a function of sample size, varying from $0.1\mu s$ to $100\mu s$. The curve has a relative steep negative slope for sample size less than $10\mu s$, and levels off thereafter. In other words, the sampling accuracy becomes worse when sample size is smaller than $10\mu s$, and does not improve much as sample size is bigger than $10\mu s$. Increasing sample size also leads to slower simulation speed. We therefore pick $10\mu s$ as our sample size for THS.

6.2. Warmup

Warmup of hardware structures has a significant impact on simulation speed. For example, in the experiments of Section 6.1, although the FDS is less than 1%, simulation speedup is only $4.72\times$ compared to detailed simulation. We want to achieve much higher speedup without sacrificing accuracy too much. We therefore explore a number of warmup strategies as described in Section 4.2.

Figure 8 plots simulation speedup as a function of the average absolute execution time prediction error for the different warmup strategies. W_0 (no warmup) leads to the highest error of 8.52% while yielding the highest simulation speedup (approximately $60\times$). Adding a small warmup (W_1) makes a great improvement, decreasing the error to 5.27%. W_9 yields another big improvement: the error comes down to 4.02%. Adding more warmup does not improve accuracy much, while negatively affecting simulation speed. In particular, W_{full} provides a $4.72\times$ speedup over detailed simulation with an error of 3.67%. We find W_9 to be a good trade-off in accuracy (4.03% average absolute prediction error) versus simulation speed ($39.72\times$). We use this warmup strategy throughout the article unless mentioned otherwise.

Fig. 9. The absolute execution time prediction error for THS under the W_{full} and W_9 warmup strategies.

Fig. 10. The absolute accuracy of THS for predicting miss rates: L1 data (instruction) cache, L2 cache, L3 cache, instruction-TLB, data-TLB, L2 TLB, and branch predictor (compared to detailed simulation).

6.3. THS Accuracy

We now report THS accuracy for its default sampling parameter values: $D = 10\mu s$, $W = W_9$, $N = 10$, and $F = 8$.

6.3.1. Absolute Accuracy. To evaluate THS' absolute accuracy, we consider a simulated eight-core system and compare THS' reported simulation results against Sniper's detailed simulation of the entire benchmark run. The average absolute execution time prediction error equals 3.67% under full warmup (W_{full}) and 4.02% under the W_9 warmup strategy (see Figure 9). For nine of the benchmarks, we obtain an error below 4%. Three of the benchmarks have relatively higher errors (namely, ferret, swaptions, and facesim), with swaptions as the most notable example. The reason is that IPC varies at the $10\mu s$ time granularity within a fairly wide range (between 0.8 and 1.5) for these benchmarks, which complicates making accurate fast-forwarding IPC predictions. Increasing sample size to $50\mu s$ solves the problem though for the swaptions benchmark, decreasing prediction error from 11.8% to 0.5%.

THS is also accurate when it comes to predicting other microarchitectural performance metrics, including miss rates for the L1 data cache (L1_D), L1 instruction cache (L1_I), L2 cache (L2), L3 cache (L3), instruction TLB (I_TLB), data TLB (D_TLB), L2 TLB (L2_TLB), and branch predictor (Bran_Pre). Figure 10 presents these performance metrics (miss rates) for both full detailed simulation versus THS. Although we observe

Fig. 11. Evaluating THS' relative accuracy: Speedup obtained via THS versus detailed simulation as a function of core count, relative to a four-core processor.

small deviations in some cases (see, for example, vips for predicting the L1 I-cache miss rate), the vast majority of the results highlight good correspondence between sampled and detailed simulation.

6.3.2. Relative Accuracy. Although absolute accuracy (i.e., accuracy of a simulation methodology in a particular design point) is important, relative accuracy is even more important for a computer architect. Relative accuracy refers to the ability to predict relative performance differences between design points. In this section, we consider THS' accuracy for predicting relative performance differences as we increase core count, from 4 to 32 cores. Figure 11 plots normalized performance speedup (relative to a four-core system) for each benchmark. THS' accuracy is high for predicting relative performance trends: the two curves representing sampled and detailed simulation almost perfectly match. For several benchmarks, we do observe a superlinear speedup, which THS accurately estimates (see blackscholes, canneal, raytrace, and vips). For several other benchmarks, we observe (almost) linear scaling (see fluidanimate, dedup, bodytrack, and facesim). A couple other benchmarks exhibit sublinear scaling (see freqmine and streamcluster in particular, due to limited parallelism and barrier synchronization overhead, respectively). The key observation here is that THS can accurately predict relative multicore scaling behavior.

6.4. Comparison Against Existing TBS Methods

Several works presented TBS methodologies recently [Ardestani and Renau 2013; Carlson et al. 2013; Jiang et al. 2013]. We refer to these works as ESESC, Periodic, and PCantorSim, respectively, against which we now compare THS. To enable an apples-to-apples comparison, we implement all three methodologies in the same simulation infrastructure, namely, Sniper, and perform two kinds of comparisons.

In our first comparison, we configure the sampling parameters as suggested in the respective papers. For ESESC [Ardestani and Renau 2013], sample size equals $20\mu s$, with $10\mu s$ detailed (processor core) warmup and $2,000\mu s$ functional warmup between samples. Periodic sampling [Carlson et al. 2013] uses program phase analysis to determine an appropriate sample size per application. We therefore take the application-specific sampling parameters from the published paper; sample size varies between $10\mu s$ and

Fig. 12. The execution time prediction error (a) and simulation speedup (b) relative to detailed simulation for the four TBS methodologies.

Table V. Comparison between THS and the Other Three TBS Techniques with Improved Warmup (W_9)

	FDS (%)	FWS (%)	FFS (%)	Simulation speed (MIPS)	Average error (%)
THS	0.82	3.69	95.49	61.59	4.03
PCantorSim_WO	3.29	13.16	83.55	27.28	5.08
ESESC_WO	1.48	4.44	94.08	49.37	7.12
Periodic_WO	11.21	0.03	88.76	11.11	4.45

200ms, with an intersample interval between 5 and 10 times the sample size; functional warmup is enabled between samples. PCantorSim [Jiang et al. 2013] leverages fractal behavior to select samples and has only one parameter, namely, sample size, which is set to $10\mu\text{s}$. Warmup takes $10\mu\text{s}$ as well before each sample.

The experimental evaluation shows that THS outperforms the other three methodologies somewhat in terms of execution time prediction accuracy (see Figure 12(a)), but provides a significant improvement in simulation speedup relative to detailed simulation (see Figure 12(b)). More specifically, THS is slightly more accurate than Periodic, with an average absolute error of 4.03% versus 4.41%, and significantly more accurate than PCantorSim (5.34%) and ESESC (6.16%). Given the analysis provided in Section 3.3, it is not unexpected that ESESC is the least accurate given its small sample size of $20\mu\text{s}$. Periodic is more accurate as it employs a much larger sample size of tens to hundreds of milliseconds. The average simulation speedups compared to full detailed simulation are $39.7\times$, $21\times$, $4.5\times$, and $3\times$ for THS, PCantorSim, ESESC, and Periodic, respectively.

In our second comparison, we consider the same (optimized) warmup strategy for all sampling methods. Namely, we consider W_9, that is, $90\mu\text{s}$ functional warmup before each sampling unit. The “warmup optimized” TBS methods are denoted as PCantorSim_WO, ESESC_WO, and Periodic_WO, respectively, in Table V, which reports a comparison in terms of FDS, FWS (Fraction of Warmup Simulation), FFS (Fraction of Fast-forwarding Simulation), absolute simulation speed, and average error. Overall, THS still is the winner yielding the highest simulation speed (61.59MIPS) and the highest accuracy (4.03% error).

The biggest benefit from THS is arguably in terms of simulation speed. There are two main factors that determine a sampling method’s simulation speed: FDS and FWS. The fewer samples, the shorter the samples (i.e., smaller FDS) and the shorter the warmup length (i.e., smaller FWS), the bigger the simulation speed. THS performs great in both aspects. By combining periodic and cantor sampling in a two-level sampling strategy,

Table VI. Case Study Configurations

<i>Case study</i>	<i>Config_1</i>	<i>Config_2</i>
#1: <i>Issue width</i>	2	4
#2: <i>LLC replacement policy</i>	LRU	S-RRIP (M takes 2-bits)
#3: <i>Homogeneous vs. heterogeneous system</i>	Eight homogeneous cores frequency = 2.66GHz issue width = 4 ROB size = 128 L1.D cache = 32KB L2 cache = 256KB branch miss penalty = 10	Four small cores and four big cores 1.5GHz vs. 3.5GHz 2 vs. 6 64 vs. 256 16 vs. 64 128 vs. 512 8 vs. 20

THS selects relatively few small-sized sampling units, leading to the smallest FDS among all methodologies (0.82%). At the same time, because of the fractal-based sampling at the second level, the samples are clustered together, which enables relatively short warmup lengths (3.69%), as previously discussed in Section 4.2. These two factors enable THS to achieve the highest simulation speed, while yielding the highest accuracy. The high accuracy is also a result of two-level sampling, which combines the best part of periodic and cantor sampling. Note that the warmup optimization improves the simulation speed of the ESESC and Periodic methods, but also decreases their estimation accuracies (especially for ESESC). The reason is that the fast-forwarding interval between two sampling units in ESESC is quite large and the short warmup length optimized for improving speed cannot eliminate the errors caused due to the *cold-start* problem. In contrast, THS's sampling units are clustered together, leading to many very small fast-forwarding intervals. This decreases the errors caused by the *cold-start* issue, facilitating us to use shorter warmup but achieve high accuracy.

7. CASE STUDIES

As noted before, relative accuracy is usually more important than absolute accuracy for a computer architect. An essential question for a computer architect to ask before employing a sampling strategy as described in this article, or a simulation methodology in general, is: Is this simulation technique accurate enough for making design trade-offs, and steer design decisions? To answer this question, we consider three case studies and evaluate whether THS is able to make correct design decisions.

The three case studies focus on different trade-offs within a core (varying issue width), LLC (varying cache replacement policy), and at the chip level (heterogeneous vs. homogeneous multicore configuration). Table VI shows details for the two design configurations being compared. In case study #1 we vary issue width from 4 (default) to 2; in case study #2 we change the LLC replacement policy from LRU (default) to S-RRIP [Jaleel et al. 2010]; in case study #3 we change the multicore configuration from homogeneous (default) to heterogeneous. The parameters not listed in Table VI are set to the default values in Table IV.

Figure 13 shows the performance improvement of one configuration over another for each case study. Generally, all four TBS methods are able to track the performance differences well. In other words, for the vast majority of benchmarks, they can accurately pinpoint the performance benefit of one configuration over another. For example, a significant performance benefit from S-RRIP over LRU is observed for *canneal* and *facesim* in *case study #2* (see Figure 13(b)). In *case study #3* (Figure 13(c)), *ferret* and *dedup* are the only benchmarks that see a performance benefit from heterogeneity. The reason is that these benchmarks employ a pipeline-parallel execution model, whereas the other 10 benchmarks use a data-parallel execution model. Threads of the

Fig. 13. Case studies evaluating the ability of TBS methods to steer design decisions: (a) core variation (issue width), (b) LLC variation (replacement policy), and (c) chip-level variation (homogeneous versus heterogeneous multicore).

Table VII. The Absolute Performance Improvement Differences between Detailed and TBS Sampled Simulations

	Average absolute performance improvement differences		
	Case #1 Issue width	Case #2 LLC replac. polic.	Case #3 Homoge. vs. Heteroge.
THS	2.9%	0.9%	2.0%
PCantorSim_WO	3.9%	1.2%	2.7%
ESESC.WO	4.2%	1.5%	3.3%
Periodic.WO	3.1%	0.9%	2.2%

benchmarks with data-parallel model are homogeneous, and the slowest threads running on the small cores in heterogeneous system determine overall performance. Hence the homogeneous system is the winner for the ten benchmarks. However, for *ferret* and *dedup* with the pipeline-parallel execution model, threads are heterogeneous, and the threads having the heaviest tasks are the critical ones to determine the final performance. In our heterogeneous system, these critical threads are pinned to big cores, hence the heterogeneous system is the winner for *ferret* and *dedup*.

Table VII shows the average absolute differences between full detailed simulation and the four TBS techniques. The average absolute difference for THS is less than 3% across the three case studies, which is slightly more accurate than *Periodic_WO*, and significantly more accurate than *Pcantorsim_WO* and *ESESC_WO* (up to 4%). Since the relative accuracy is so important, our suggestion for architects is that if a design's average improvement is quite small (e.g., less than 3%–5%), detailed simulation should be performed to discern the accurate performance delta. Also, THS provides a reliable and very fast evaluation method, which makes it possible for architects to explore a

huge design space, and identify which design is the winner. However, if one wants to know the exact number of an enhancement, full detailed simulation is still necessary.

8. RELATED WORK

We now briefly cover related work for completeness.

Single-Threaded Application Sampling. Sampled simulation of single-threaded applications is widely used [Yi et al. 2005]. Conte et al. [1996] were the first to apply sampling theory to processor simulation. Sherwood et al. [2002] leverage phase analysis to identify a limited number of representative, and relatively long sampling units (100 million instructions). Wunderlich et al. [2003] introduce periodic sampling with statistical bounds across a large number of relatively small sampling units (1,000 instructions). Yu et al. [2010] guide the selection of sampling units using fractal theory. Van Biesbrouck et al. [2004] sample representative multiprogram workload simulation points. COTSon [Argollo et al. 2009] dynamically selects samples guided by changes in executed code. All of these sampling approaches specify their sampling parameters in terms of instructions—IBS—which does not apply to multithreaded applications, as extensively argued in this article.

Multithreaded Application Sampling. Ardestani and Renau [2013] and Carlson et al. [2013] introduce TBS, which guides sample selection based on execution time rather than instruction count. Both approaches employ periodic sampling. Cantor sampling proposed by Jiang et al. [2013] selects sampling units based on cantor fractal theory. In this article, we analyze both periodic and cantor sampling in great detail, and reveal a number of novel and surprising insights, which ultimately leads to THS, combining the best of both worlds.

Sampled simulation techniques have been proposed for particular classes of workloads. Ekman and Stenström [2005] target multithreaded workloads with independent, nonsynchronizing threads; similarly, Wenisch et al. [2006] target commercial workloads in which threads do not interact. Perelman et al. [2006] leverage application phase analysis to estimate multithreaded application IPC, not overall execution time. Bryan et al. [2012] and Carlson et al. [2014] focus on barrier-synchronized applications. THS applies to a broader set of multithreaded applications while achieving high accuracy and simulation speed.

9. CONCLUSION

This article first comprehensively analyzes existing sampling schemes in the context of TBS, and reveals a number of novel, useful insights, some of which are surprising: (1) state-of-the-art TBS approaches accurately predict sample IPC; (2) accurately predicting fast-forwarding IPC is substantially more important to TBS' overall accuracy than accurately predicting sample IPC; (3) fast-forwarding IPC estimation accuracy is determined by both the sampling unit distribution and the fast-forwarding IPC estimation method; (4) periodic sampling is more accurate at coarse-grain time scales because of the uniform distribution of sampling units; (5) cantor sampling is more accurate at fine-grain time scales because of its ability to more accurately predict fast-forwarding IPC at variable, small time scales; and (6) random sampling is not appropriate for TBS due to variations across different runs.

Motivated by these insights, we propose THS as a solution for the simulation challenge of multithreaded applications. THS combines the best of both worlds by selecting coarse-grain sampling units periodically (first level) before selecting fine-grain sampling units within coarse-grain sampling units following cantor fractal theory (second level). The fact that fine-grain sampling units are clustered leads to reduced warmup requirements, yielding even higher simulation speeds. Overall, we find THS to

significantly outperform state-of-the-art in TBS in both accuracy and simulation speed, yielding an average absolute execution time prediction error of 4% at a $40\times$ simulation speedup compared to detailed simulation of an eight-core system running PARSEC benchmarks. In addition, we demonstrate that THS is accurate across the design space (good relative accuracy), and enables making correct design decisions in practical design studies.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable and constructive comments.

REFERENCES

- A. R. Alameldeen and D. A. Wood. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro* 26, 4 (2006), 8–17.
- E. K. Ardestani and J. Renau. 2013. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 448–459.
- E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. 2009. COTSon: Infrastructure for full system simulation. *SIGOPS Operating System Review* 43, 1 (Jan. 2009), 52–61.
- C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.
- N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.
- P. Bose and T. M. Conte. 1998. Performance analysis and its impact on design. *IEEE Computer* 31, 5 (May 1998), 41–49.
- P. D. Bryan, J. A. Poovey, J. G. Beu, and T. M. Conte. 2012. Accelerating multi-threaded application simulation through barrier-interleaved parallelism. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 117–126.
- T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.
- T. E. Carlson, W. Heirman, and L. Eeckhout. 2013. Sampled simulation of multi-threaded applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.
- T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. 2014. BarrierPoint: Sampled simulation of multi-threaded applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.
- J. Chen, L. K. Dabbiru, D. Wong, M. Annavaram, and M. Dubois. 2010. Adaptive and speculative slack simulations of CMPs on CMPs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 523–534.
- T. M. Conte, M. A. Hirsch, and K. N. Menezes. 1996. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD)*. 468–477.
- E. Duesterwald, C. Cascaval, and S. Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 220–231.
- M. Ekman and P. Stenström. 2005. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 89–99.
- 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. 2002. Asim: A performance model framework. *IEEE Computer* 35, 2 (Feb. 2002), 68–76.
- A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 60–71.

- C. Jiang, Z. Yu, H. Jin, C. Xu, L. Eeckhout, W. Heirman, T. E. Carlson, and X. Liao. 2013. PCantorSim: Accelerating parallel architecture simulation through fractal-based sampling. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (Dec. 2013), Article 49.
- L. K. John. 2006. Aggregating performance metrics over a benchmark suite. In *Performance Evaluation and Benchmarking*, L. K. John and L. Eeckhout (Eds.). CRC Press, 47–58.
- J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 1–12.
- A. Patel, F. Afram, S. Chen, and K. Ghose. 2011. MARSSx86: A full system simulator for x86 CPUs. In *Proceedings of the 48th Design Automation Conference (DAC)*. 1050–1055.
- H. O. Peitgen, H. Jurgens, and D. Saupe. 2004. *Chaos and Fractals: New Frontiers of Science*. Springer.
- E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings of the International Conference on Parallel and Distributed Processing Symposium (IPDPS)*.
- S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. 1993. The Wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 48–60.
- D. Sanchez and C. Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 475–486.
- T. Sherwood, E. Perelman, and B. Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 3–14.
- T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 45–57.
- R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 335–344.
- V. Uzelac and A. Milenkovic. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 207–217.
- M. Van Biesbrouck, T. Sherwood, and B. Calder. 2004. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 45–56.
- T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. 2006. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro* 26, 4 (July 2006), 18–31.
- R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 84–97.
- J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. 2006. The future of simulation: A field of dreams. *Computer* 39, 11 (2006), 22–29.
- J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. 2005. Characterizing and comparing prevailing simulation techniques. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 266–277.
- Z. Yu, H. Jin, J. Chen, and L. K. John. 2010. CantorSim: Simplifying acceleration of micro-architecture simulations. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'10)*.

Received March 2015; revised July 2015; accepted August 2015