ESESC: A Fast Multicore Simulator Using Time-Based Sampling

Ehsan K. Ardestani and Jose Renau
Dept. of Computer Engineering, University of California Santa Cruz

Abstract

Architects rely on simulation in their exploration of the design space. However, slow simulation speed caps their productivity and limits the depth of their exploration. Sampling has been a commonly used remedy. While sampling is shown to be an effective technique for single core processors, its application has been limited to simulation of multiprogram, throughput applications only. This work presents Time-Based Sampling (TBS), a framework that is the first to enable sampling in simulation of multicore processors with virtually no limitation in terms of application type (multiprogrammed or multithreaded), number of cores, homogeneity or heterogeneity of the simulated configuration (4.99% error averaged across all the evaluated configurations). TBS also is the first to enable integrated power and temperature evaluation in statistically sampled simulation of multicore systems (with 5.5% and 2.4% error on average, respectively). We implement an architectural simulator based on TBS, called ESESC, that provides a holistic set of tools for a fair evaluation of different architectures.

1. Introduction

Exploration of the design space in computer architecture heavily relies upon simulation. Prohibitively slow timing simulation has motivated emergence of various techniques to speed up the exploration process. Sampling is among the most effective techniques adopted to mitigate the problem. It is a rigorous methodology which reduces the simulation time by selecting a small, yet representative subset of the execution trace for detailed timing simulation.

While sampling has shown a great potential for single threaded or multiprogrammed applications, there are no sampling models for multithreaded application [1] simulations nor for multicore temperature simulations. The goal of this work is to provide a sampling framework to quickly explore architectural performance, power, and temperature trade-offs in multicore homo- and heterogeneous configurations running multithreaded applications.

[1]Throughout this paper, we use the term multiprogram to refer to multithreaded applications that do not use shared memory to communicate and synchronize, such as SPEC-rate, while multithreaded term is used for applications that use shared memory for communication such as PARSEC.

For single core systems, Wunderlich *et al.* [31] apply the statistical sampling theory to architectural simulation (SMARTS). Sampling becomes more challenging for multicore configurations due to the following related reasons: IPC is no longer a valid metric to evaluate the results [3], the execution exhibits variations in the dynamic stream of instructions compared to single threaded execution [2], it could simulate a non-representative overlap of the thread's execution, and the progress of temperature and therefore, its performance implications are unclear.

The first challenge in simulation of multicore systems running multithreaded applications is the evaluation of results. After varying different architectural parameters, a precise comparison should enable a fair evaluation of architectural trade-offs. The performance of a single threaded application can be compared across different architectures using IPC. The same metric also can be used to evaluate the speedup in a multicore system running multiprogram applications (*e.g.*, running *gzip* and *crafty* together). This is because the assumption of constant instructions per program could remain valid for these type of applications (*e.g.*, by excluding system instructions). However, IPC falls short to faithfully compare multicore configurations running multithreaded applications [3].

In multithreaded applications, threads can be executing instructions that do not contribute to progress of the program. Nonetheless, these instructions count toward the overall IPC (*e.g.*, polling to acquire a lock, but a higher IPC because of that does not mean speedup!). A more fundamental and problematic issue is that an increased performance for one thread does not necessarily contribute to shortening the execution time of the program's critical path. Instead of IPC, *execution time* [2] is the ultimate metric to compare different configurations' relative speed [3, 7, 8, 18].

In a sampled simulation, timing is only modeled within the samples, while the intervals between samples are fast-forwarded with functional emulation or warmup that lacks any timing information. It results in the collapse of the program execution time. Hence, the application of sampling has been limited to single thread or multiprogram applications, which can rely on the IPC for the evaluation.

Sampling could also distort the overlap of threads and their interference. The consistency of progressed time is not

[2]We use the term *progressed time* as well to refer to the execution time of a program while it is making progress.

maintained in fast-forwarded intervals, and the progress of threads could start diverging with regard to the un-sampled simulation. This could potentially result in simulating non-representative overlaps of the execution, which elevates non-deterministic and non-representative utilization of the shared resources. These effects are exacerbated in a heterogeneous configuration.

Wenisch *et al.* [28] extend SMARTS to multicore simulation for throughput applications. However, the sampling for multithreaded applications in a shared memory model remains an open problem [29].

Modern processors adapt to temperature responses. Equally important, temperature has an exponential impact on leakage power. Hence, simulation of power and temperature effects, as pertinent design parameters, are desired. Since the overall IPC does not reflect the execution time in multithreaded programs, Energy Per Instruction (EPI) can no longer be used to compare energy efficiency in sampled multicore simulation. To the best of our knowledge, there is no work proposed to enable energy comparison and simulation of temperature in a statistically sampled multicore simulation.

This paper is the first to introduce a sampling-based simulation methodology that enables rapid evaluation of virtually any multicore configuration, independent of application's type (multithreaded, multiprogram or a mixture of both), core count, and heterogeneity of the configuration. The proposed sampling framework, called Time-Based Sampling (*TBS*), samples the execution regarding progressed time rather than instruction count. All the aforementioned challenges are addressed, and the accuracy of the proposed method is evaluated by running a combination of SPEC CPU2000 [14], CPU 2006 [13], Parsec [6] and SPLASH-2 [30] benchmarks on different homo- and heterogeneous configurations.

Our open source implementation of *TBS*, called enhanced SESC [21] or *ESESC*, shows that the proposed method provides robust estimation of the speedup for multithreaded applications with average error of 4.99% across all the evaluated multicore configurations, compared against full timing simulations (maximum error 10.6% for radix on a 4-core heterogeneous processor). This paper is also the first to propose a framework that enables thermal simulation for such configurations as well (with 5.5% and 2.4% error on average for power and maximum temperature respectively).

2. Related Work

Sampling for single core: Sampling in architectural simulation has been studied in different works [31, 22]. Wunderlich *et al.* [31] apply the statistical sampling theory to architectural simulation (SMARTS). During the program execution, a small set of instructions are periodically sampled for simulation to capture the accumulative statistics. Each sample is thousands of instructions long. Hence, a warm up period is needed to populate the microarchitectural states and avoid measurement errors due to cold start. In our work, we categorize SMARTS as Instruction-Based Sampling (*IBS*). Sherwood *et al.* [22] propose a phase-based sampling based on Basic Block Vectors (BBV). In this work, our focus is on the statistically sampled simulation.

Temperature simulation with sampling: Extra consideration needs to be taken into account for thermal evaluation in a sampled simulation. Coskun *et al.* [10] use SimPoint to sample performance and power phases, and reuse them to reconstruct the whole power trace on which thermal computation is performed. Ardestani *et al.* [4] extend the sampling to the thermal domain by reconstructing a power-time trace as the simulation progresses. To the best of our knowledge, *TBS* is the first to enable thermal evaluation for statistically sampled simulation of shared memory systems running multithreaded applications.

Accelerating sampling: Different techniques are proposed to accelerate a sampling-based simulation such as native or hardware-accelerated methods(*e.g.*, [9]), checkpoint based methods (*e.g.*, [27]), or limited warmup (*e.g.*, [11]). We implemented BLRL [11] but it did not reduce the simulation time in our setup (memory warmup is very fast in our setup, and smaller intervals reduce the speed of functional emulation, thereby diminishing the benefit). Ekman *et al.* [12] use a matched-pair technique to reduce the number of samples comparing different architectures running multiprogram applications. Our proposed sampling framework provides accurate results independent of the application type.

Sampling for multicore: Wenisch *et al.* [28] developed SimFlex for multicore throughput applications based on statistical sampling. The main consideration in multicore domain is to increase the sample length. SMARTS specifies the sample length regarding the number of instructions, while in SimFlex it is recognized that cycle-based samples provide more stable results. However, the samples are distributed based on the instruction count. Van Biesbrouck *et al.* [25] introduce Co-Phase matrix to enable multicore evaluation for multiprogram applications using phase-based sampling (SimPoint). This method does not scale well, because the co-phase matrix dimensions grow by number of cores and dynamic architectural states of them (*e.g.*, DVFS states). *TBS* is the first to propose a statistical sampling framework for multithreaded applications.

Complementary techniques: Parallelization of timing simulation provides an opportunity to trade accuracy for simulation speed by breaking the cycle-by-cycle synchronization among the threads [8, 18, 7]. As a result, the simulation speed could potentially scale with the number of cores in the simulated configuration. These techniques are orthogonal to the effect of sampling in reducing simulation time.

3. Sampling Methodology

The collapse of progressed time in a sampled simulation brings up two issues: 1) the evaluation relies on IPC rather than execution time; 2) it leads to inconsistency in progress among threads, which could in turn result in simulation of non-representative overlap of threads' execution. These two challenges limit the application of sampling to single threaded or multiprogram applications. For a sampling framework to support multithreaded applications, these two challenges need to be overcome.

3.1. Sampling: Time vs. Instructions

Existing statistical sampling methods (*e.g.*, SMARTS [31]) specify the sampling parameters, such as the sample length and/or the interval between samples, in terms of the number of instructions. We call the SMARTS-like sampling Instruction-Based Sampling or *IBS* for short.

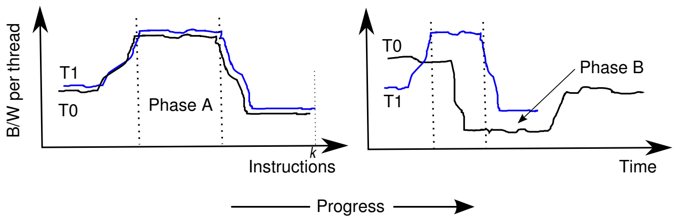


Figure 1: Progress regarding the number of executed instructions (left) and time (right). The y-axis is an arbitrary execution metric such as bandwidth demanded by each thread.

Figure 1 shows the execution of k instructions for two sample threads, $T0$ and $T1$, with average IPC of X and Y respectively, where $X < Y$. The figure on the left shows the progress with respect to the instructions. The plot shows how different phases of threads overlap one another. The figure on the right, on the other hand, shows the progress with respect to time. It shows the actual overlap of these two threads, which is different from the one observed through progress of instructions. Even in a homogeneous system, not all the threads run at the same IPC rate, and as a result the actual overlap is different from the overlap observed through instruction progress. This non-representative overlap can result in significant differences in the reported metrics.

IBS has no mechanism to avoid divergence of progress among threads (except for synchronizations in the application, which we will explain in Section 3.1.1). Threads with different IPC could quickly diverge in their progress. This could create an unrealistic interference among threads that introduces variation in the execution due to sampling that *does not exist in the original execution.*

To illustrate this effect, we simulate a synthetic multithreaded kernel. Figure 2 shows its functional diagram. It has a worker thread that cycles through two phases. *Phase A*, which is a bandwidth-bound phase that copies a buffer into thread's local buffer. *Phase B* is a high IPC and compute-bound phase. Performance of the kernel is sensitive to the overlap of the phases. *Phase A* pressures the memory subsystem, and demands higher bandwidth, and the overlap of

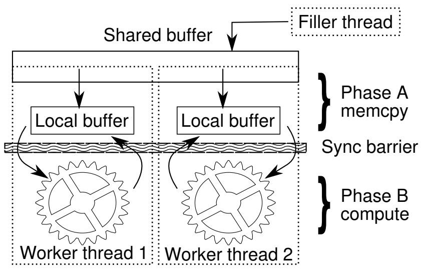


Figure 2: Functional diagram of a synthetic kernel with two distinctive phases.

two *Phase A*s could degrade the performance. There is a barrier at the end of *Phase A* to synchronize the threads.

We run *IBS* and an un-sampled full simulation for a heterogeneous processor. As expected, the sampled and full simulation match for a single worker thread. However, the sampled simulation shows higher error for the configuration with two worker threads. In particular, the memory system statistics, such as cache miss rates on different levels, shows up to 100% error due to non-representative overlap of the threads. Another observation is that the sampled simulation shows much higher variations across the runs (2.5X standard deviation of the full simulation). This is because the overlap observed by *IBS* is not deterministic, and could change across simulation runs. This adds to the actual variation observed by the full simulation. If we define the sampling parameters regarding time (*TBS*, explained in Section 3.2), the error in the memory statistics and variation drop to 17% and 0.2X respectively.

3.1.1 Synchronization:

Multithreaded applications use synchronization to provide safe communication and maintain control among threads. Different synchronization primitives (*e.g.*, locks, barriers, and conditions) can be divided into primitives that either *spin* or *sleep* while trying to meet a condition.

In *spin* primitives, a thread actively executes instructions while waiting to acquire the lock or associated condition. The spinning generates a nondeterministic stream of instructions that causes variations across runs. In *sleep* primitives, the waiting thread is unscheduled by the OS. We observe that this results in a more stable dynamic stream of instructions, and less variations of the dynamic program behavior across the runs in the user mode.

The divergence increases even further in a heterogeneous system. Figure 3 shows a histogram of divergence. The y-axis shows how frequent threads in an *IBS* multithreaded simulation diverge in their progressed time The x-axis indicates the magnitude of divergence that has happened. We run a number of Parsec benchmarks, and reconstruct and record the divergence of progressed time among threads at each sample (experiments ran on a 4-core heterogeneous processor. See Table 6). We use the same reconstruction proposed by [4] for single core configurations. In some ap-

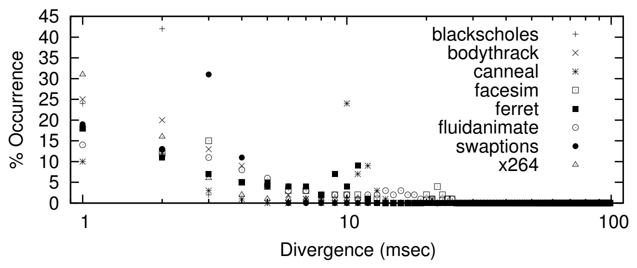


Figure 3: Histogram of divergence of execution time in a sampled simulation.

plications, the threads are almost identical; perform around the same amount of work without contention over the shared resources; and use barriers to synchronize. *Blackscholes* is an example of such applications. In *blackscholes*, the threads progress with the same rate and the divergence is limited. Other applications such as *ferret* and *fluidanimnate* do not use global barriers (locks and condition synchronization primitives are used though), and provide higher probability of divergence in the progress of different threads. This divergence is not part of the program behavior, rather a side effect of the sampling. In this figure, 27% of the samples have a divergence greater than 10*ms*. The divergence also happens in a homogeneous processor (22% in the baseline configuration, Table 6).

While we do not observe an adverse effect on the result of running multiworkload applications, for multithreaded configurations with applications exhibiting significant sharing and contention patterns, it seems necessary to define the sampling parameters with respect to progress in time rather than the number of instructions to avoid divergence of the progress among threads. In Section 3.3 we will explain that this is also necessary to enable temperature simulation across a chip.

3.2. TBS: A Time-Based Sampling

To maintain the progressed time, and avoid divergence of progress among threads and preserve the overlap of execution among them in a sampled simulation, we introduce Time-Based Sampling(*TBS*). Similar to *IBS*, *TBS* breaks the sampling intervals into 3 subintervals: Functional fast forwarding that performs Memory system Warmpup (*MW*); Detailed Warmup (*DW*) that performs detailed timing to warm up the pipeline, but discards the statistics; and Detailed Timing (*DT*). The samples are gathered during *DT*. However, in *TBS* sampling subintervals are defined to have a fixed length in number of cycles [3], as shown in Figure 4. In contrast, subintervals have fixed number of instructions in *IBS*. Unlike *IBS*, as *TBS* progresses by executing through the sampling subintervals, the execution time of the application is available by adding up the predefined length of subintervals.

[3]We use time and cycles interchangeably in this context, $Time = \frac{Cycles}{freq}$.

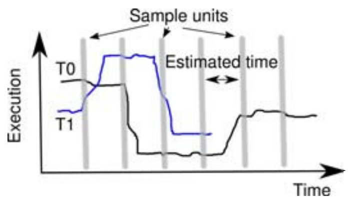


Figure 4: Periodic *TBS* Sampling. Sampling intervals have fixed length in terms of number of cycles.

Let $inst_s$ be the length of subintervals at runtime in terms of instructions, and cyc_s be the length in terms of number of cycles. Table 1 shows the property of each interval for *TBS* and *IBS*.

Method	$inst_s$	cyc_s	Description
IBS	Constant	Variable	Progressed time diverges among threads
TBS	Variable	Constant	Threads' progressed time advances synchronized as it would in full simulation

Table 1: Different properties for each sampling method.

Simulation eventually comes down to the execution of instructions. *TBS* adjusts the number of instructions at runtime to keep the length for each subinterval, in terms of time, constant. Therefore, the length is converted at runtime, from number of cycles to the number of instructions based on the observed performance. This is done independently for each thread. Equation 1 is the conversion formula.

$$\#inst_{(i,s)} = \#cyc_s \times IPC_i,\ i = (1..n),\ s : (MW, DW, DT) \quad (1)$$

$\#inst_{(i,s)}$ is the number of instructions for subinterval s of the i^{th} sampling interval. $\#cyc_s$ is the length of subinterval s in terms of number of cycles. It is a sampling parameter and is predefined for each subinterval type (See Section 3.2.2, and Table 4). IPC_i is the runtime performance of interval i. n is the total number of samples.

As an example, for a fast-forward subinterval MW_i, the IPC_i is predicted at runtime. The length of the subinterval has been predefined as a fixed number of cycles. So, the conversion simply lets us know how many instructions need to be fast-forwarded for that particular interval to make up for the predefined length in terms of time.

3.2.1 IPC Prediction:

Now the main question is how the performance of each subinterval (IPC_i) is obtained. For the *DT* subinterval the answer is simply available, because detailed timing simulation is performed. Other subintervals, however, by design do not perform detailed timing modeling nor do they maintain the statistics. For such cases, we predict the IPC to figure out the number of instructions to be executed (as an example of related work predicting IPC see [4]).

The statistical variations among consecutive samples are bounded by the sampling parameters selected, which is done based on the observed variation in the program (Discussed more in Section 3.2.2). This makes the accurate prediction of IPC based on previously observed intervals possible.

We evaluate 3 different prediction methods. For reference, the *Naive* method assumes IPC of 1 for all the subintervals between samples. *Last* method uses the last measured IPC. Inspired by the effects in thermal stage (*e.g.*, exponential decay of thermal effects), *WMA* method uses a weighted average of last h (3 to 5) samples. The most recent sample has the highest weight, and the last h^{th} sample has the lowest. Table 2 summarizes these methods.

Method	Description
Naive	Assumes IPC=1
Last	IPC of the last sample
WMA	**W**eighted **M**oving **A**verage of last 3-5 samples: $EIPC_i = \frac{\sum_{k=0}^{h-1} \frac{1}{2^k} \times IPC_{i-k}}{\sum_{k=0}^{h-1} \frac{1}{2^k}}$

Table 2: IPC prediction methods for fast-forwarded intervals.

We run a set of benchmarks in full, and compare the predicted IPC against the observed IPC, for all the fast-forwarded subintervals (We used parameters presented in Table 4 for the size of subintervals). Figure 5 summarizes the accuracy of each prediction method. The *Naive* prediction performs worst with average error of 17%. The deviation of errors is also very high (Min and Max error of 5% and 37% respectively). The *Last* and *WMA* predictions perform best with average error of around 8%. The deviation of errors is much less, which means the prediction also captures the transients in the IPC (We tried WMA with no weighting but it did not perform as well). While either of *Last* or *WMA* methods can be used, we use the *WMA* as it provides more stable predictions in terms of standard deviation, resulting in more accurate reconstruction of performance trace for thermal computation (Explained in Section 3.3). This confirms that IPC of the fast-forwarded intervals can be predicted using previously observed samples. The overall IPC error for the program is even less than the prediction error for the transient IPCs as the errors can cancel out each other through the execution.

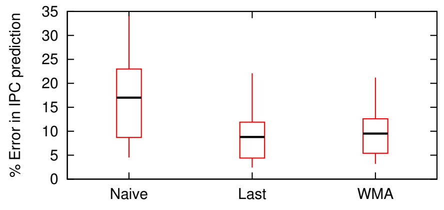


Figure 5: Error in prediction of IPC.

3.2.2 Sampling Parameter Selection:

As evaluated in [31] and [28], selection of sampling parameters depends on the coefficient of variation of samples ($\hat{V} = \frac{\sigma}{\mu}$) in the execution, and the desired confidence level ($1-\alpha$). The number of samples can be obtained by $n \geq [(z \cdot \hat{V}).\varepsilon]^2$, where $z = 100(1-\alpha/2)$, and $\varepsilon \cdot \bar{X}$ is the confidence interval. Also the impact of detailed warmup (*DW*) has to be taken into account as an interdependent factor in determining the length of samples in detailed timing (*DT*).

This process is discussed in detail in SMARTS [31]. Hence we do not repeat the procedure, and refer to this related work for details ([28] also follows the same procedure). The only difference is that for *TBS*, *IPC* is the metric for which the coefficient of variability is studied . This is because the number of cycles is the constant denominator. In SMARTS (or *IBS* in general), the denominator is number of instructions (*CPI*).

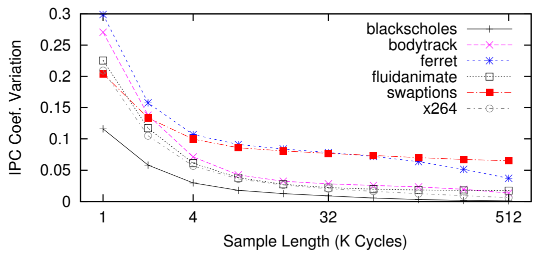


Figure 6: Coefficient of variation ($\hat{V}$) of IPC for different sample length in multithreaded benchmarks.

Figure 6 shows the coefficient of variation in IPC over different sampling sizes (*DT*). Selecting a larger sampling length results in smaller $\hat{V}$, and less number of samples. However, based on Figure 6, sampling length of around 32K is a point of diminishing return, as longer samples do not reduce $\hat{V}$, but will keep reducing the simulation speed ([28] reports the same trend for throughput applications).

For the best results, the parameters can be selected per benchmark. We try to select a set of parameters that work well with all the benchmarks. This comes at the cost of slightly slower simulation. We select the sample length of 50K and the detailed warmup of 20K. Table 4 summarizes the selected parameters.

3.2.3 Synchronization Among Samples:

TBS keeps the progress of samples synchronized. All threads start a sampling interval by *MW* and finish by *DT*. Unlike *IBS*, all thread should finish executing a sample around the same time. Should a thread finish the simulation of the current sample (*DT*) faster due to IPC prediction error, it will switch to *DW*, which is detailed timing without gathering statistics. This ensures all the active threads maintain their pressure on the shared resources as long as the current sample is being processed. Once all threads have finished

DT, they start fast-forwarding to the next sample. The fast-forward for threads executing instructions in *DW* waiting for other threads will be adjusted based on how long each of them has been waiting in *DW*.

While spin-lock loops do not contribute to the actual progress of the application (which renders IPC metric misleading), they do contribute to the progressed time, and thus their appearance in any interval, including fast-forwarding, will not result in inconsistency.

3.3. Multicore Thermal Simulation

Performing realtime performance, power, and temperature simulation enables exploration of temperature- and power-aware adaptations, which is an integrated feature of modern processors. Modeling the thermal progression depends on two run time factors: power and time. Sampling introduces complexity in temperature simulation due to the collapse of progressed time. Ardestani *et al.* [4] propose a reconstruction-based method to model temperature in a sampled simulation. However, as we discussed in Section 3.1, the progressed time of the threads diverges in a sampled multicore simulation. Therefore, the reconstruction of progress time for each thread would not suffice for multicore thermal simulation.

If the consistency of the progressed time is lost, the realtime thermal computation becomes impossible. Thermal computation can be performed with a lag compared to the progress of performance domain in a checkpoint style. However, realtime interference of temperature and performance would be compromised. As a result, *IBS* methods, including [4], fail to simulate the progress of temperature in a multicore configuration.

TBS ensures the consistency of progressed time for all threads in a sampled simulation. Therefore, all threads are in the same progressed time at the end of each sample. This makes a reconstruction-based thermal simulation for multicore possible. Similar to [4], we use a formulation similar to the *WMA* in Table 2 (replace IPC for Power) to estimate the power consumption of the fast-forwarded intervals based on the measured powers of last h samples (error within 6%). The weighting resembles the exponential decay of thermal effects due to thermal time constant. While both *Last* and *WMA* provide accurate predictions, we observe that *WMA* results in a slightly more accurate thermal trace. Knowing the length of the last interval, and its power consumption, the power-time trace is reconstructed, and the temperature progress of the chip is computed for that interval. This is the first time that a statistically sampled simulation can also simulate temperature transients for a multicore configuration.

3.4. Evaluation Methodology

In the case benchmarks are run to completion, the evaluation of the results is fairly simple and feasible by comparing the execution times. However, unlike multithreaded applications, multiprogram applications finish at different times, which makes it harder to mark a fair end to terminate the simulation. The same is true for partially simulating these applications. Vera *et al.* [26] discuss fairness in the evaluation of multicore systems, and similar to Tuck *et al.* [24] propose a re-execution based evaluation.

To have a fair evaluation, we propose that the user 1) select a stable region of interest (ROI); 2) apply TBS over the selected region; 3) use execution time to evaluate the results. Marking ROI for multithreaded applications is a common practice. For single thread or multiprogram applications, ROI can be defined as a constant number of instructions, and observed by counting instructions during the emulation or sampling, or by marking the binary.

The re-execution mechanism is necessary to maintain fairness in evaluation of multiprogram applications [26]. Should any of SPEC threads in the multiprogram applications finish the ROI (or reach program finish before others) it will continue execution (or will rerun), but the statistics outside of the ROI (or after rerun) will be discarded for that particular thread. This is to keep the pressure on the shared resources, until all the threads finish the ROI.

We run all multithreaded applications to completion. Hence the evaluation is simply carried out by comparing the execution times.

One possible problem with *TBS* however, is that if the number of instructions in a sample becomes very small, it could introduce extra error due to the limitations of the timing simulation. This situation could happen for example in case of a sample with catastrophic slow performance. Note that if these slow regions are frequent, thus statistically important, the sampling parameter selection procedure should capture the effect accordingly.

There are different metrics proposed to report the speedup running multiprogram applications, compared to a single program execution. [23] used weighted speedup, and a harmonic mean is used in [16]. This work, however, does not study the speedup of an isolated run of a single application against a multiprogram run. We compare across different multicore configurations, and we compute the aggregated IPC of cores ($IPC = \sum^{n} \#Inst / \sum^{n} \#Cycles, n = \#cores$) and scale it by the number of active cores to get the speedup.

4. Experiment Setup

We modify SESC [21], and use QEMU [5] as the functional emulator executing ARM instructions. The simulator offers 4 different execution modes: *R* for functional emulation only (Rabbit), *MW* warms up memory subsystem, *DW* performs detailed warmup through timing simulation, but the statistics are discarded, and finally *DT* in which the detailed timing simulation is performed. We use a modified version of McPAT [15] for power. A modified version of SESC-Therm [19] is used to compute temperature and scale leakage power consumption accordingly. We use a model similar to [17] to scale leakage based on temperature. The resulting simulator is called Enhanced SESC (*ESESC*). Table 3 and Table 4 show the sampling methods and parameters.

Method	Description
Full	Full timing simulation
IBS	SMARTS-like simulation
TBS	Time-based sampling simulation

Table 3: Simulation methods.

Parameter	*IBS*	*TBS*
MW	496e4	493e4
DW	1e4	2e4
DT	3e4	5e4
History Size (in WMA)	-	5

Table 4: Sampling parameters. The unit for the sampling modes is instructions for *IBS* and cycles for *TBS*. The unit for History Size is the number of samples.

4.1. Evaluated Architectures

We configure 4 different CMP systems to evaluate the proposed framework. To run the experiments for different core counts as well as different core configurations, we use a fast (F) and a slow (S) configuration to build up the CMPs. Table 5 lists the architectural parameters. Configurations are labeled with a $/(CT)+/$ format to specify their core count followed by the core type. For example $4F$ is a 4-core CMP configured with the fast cores, and $2F2S$ is a 4-core heterogeneous configuration with 2 fast and 2 slow cores. Table 6 summarizes the configurations.

Parameter	Fast (F)	Slow (S)
Freq	3.0 GHz	
I$	32KB 2w (2c hit) private	
D$	32KB 8w (3c hit) private	
L2	256KB 16w (12c hit) private	
L3	4MB 16w (12c hit) shared	
Coherence	MESI	
Mem.	180 cyc	
BPred.	10 tab. ogehl 76Kb	Hybrid 38Kb
Issue	4	2
ROB	256	56
IWin.	32	16
Load/StoreQ	48/32	16/8
Reg(I/F)	128/128	80/64

Table 5: Architectural parameters

Config	Description
Homogeneous	4F (baseline), 8S
Heterogeneous	2F2S , 4S4F

Table 6: Different configurations used in the experiments. See Table 5 for more detail on F and S cores.

4.2. Applications

Throughout this paper, we use the term multiprogram to refer to the multithreaded applications that do not use shared memory to communicate, while the multithreaded term is used for applications that communicate through the shared memory and use synchronization. We use applications from SPEC CPU 2000 [14], CPU 2006 [13], Parsec [6] and SPLASH-2 [30] benchmarks suits. A random combination of SPEC applications represents the multiprogram category of applications. The pool contains 29 single threaded applications. 8 Parsec (the simlarge input set) and 4 SPLASH-2 applications represent the multithreaded class of applications. The multithreaded applications in Parsec suite demonstrate different rate of sharing and data contention from low (*e.g.*, *blackscholes*) to high (*e.g.*, $ferret$). Table 7 lists these applications. To save space, we use two characters to identify each application. All the multithreaded benchmarks are run to completion.

Suite	Applications
SPEC	**eq**uake, **gc**c, **mg**rid, **me**sa, **ar**t, **vp**r, **ap**plu, **so**plex, **tw**olf, **wu**pwise, **pe**rlbench, **sw**im, **cr**afty, **po**vray, **mi**lc, **vo**rtex, **li**bquantum, **mc**f, **ga**p, **le**slied, **bw**aves, **lb**m **as**tar, **sp**hinx, **de**alII
Parsec	**bl**ackscholes, **bo**dytrack, **ca**nneal, **fa**cesim, **fe**rret,**fl**uidanimate, sw**a**ptions, **x2**64
SPLASH-2	**oc**ean, **ff**t, **fm**m, **ra**dix

Table 7: Benchmarks

Category	Benchmark
MultiProgram	combination of SPEC applications
MiltiThreaded	Parsec or SPLASH-2
Mix	combination of MultiProgram and Multithreaded

Table 8: Evaluated application categories.

For the 8-core configurations, we also run a mixture of both multi-workload and multithreaded applications. While in the 4-core configurations the cores are fully loaded, the 8-core configuration includes benchmark combinations that load a subset of available cores. Table 8 shows the categories of applications we evaluate.

For all the experiments, we run the multithreaded applications within their scalable region. We ensure that the working set of the applications are the same, and applications themselves do not change the working set or number of threads. This is to carry out a fair comparison between different CMP configurations with different number of cores.

5. Evaluation

This Section presents evaluation results. The accuracy of performance estimation is discussed in Section 5.1. Section 5.2 evaluates the power and temperature estimations, and Section 5.3 and Section 5.4 discuss the simulation speed and scalability of the framework respectively.

5.1. Accuracy

Figure 7 presents the overall error in the reported speedup when comparing different configurations. It compares 3 application categories and 2 sampling methods, running a 4-core homogeneous as the baseline and 3 other CMP configurations (see Table 6). The error is computed comparing the reported speedup of each method to the speedup reported by execution time in the full simulation. The execution time is used for *TBS*, while *IBS* uses IPC to compute the speedup, as it lacks the progressed time information.

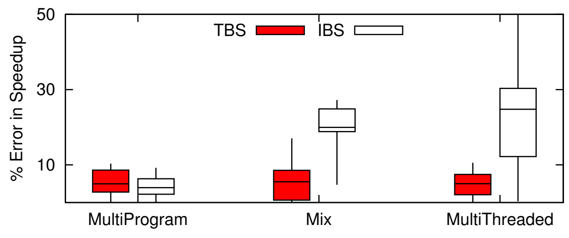


Figure 7: Error in the speedup estimation. The errors of all the configurations are grouped together.

Multiprogram workloads are deterministic without contention between threads. As a result, both IPC and execution time agree and are acceptable to estimate speedup. As expected, the multiprogram category demonstrates a low error for both *IBS* and *TBS*. In both cases, the error is within 10%.

A more challenging case is the multithreaded category. The error in the reported speedup by *IBS* increases to up to 30% and 50% for mix and multithreaded categories respectively. *TBS*, on the other hand, still provides stable accurate results across all the application categories. While we only report detailed relative speedup error, the absolute error in the execution time remains within 7%.

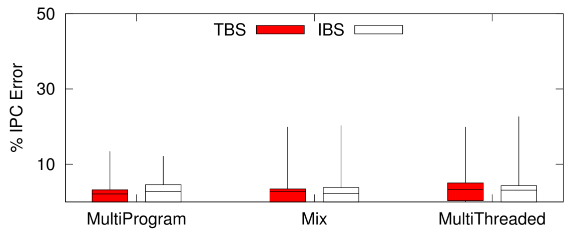


Figure 8: Error in IPC estimation. Both *IBS* and *TBS* estimate IPC accurately.

As we discussed in Section 3, IPC is not a faithful metric for comparing different configurations. For example, fft has accumulative IPC of 1.95 on the $4F$ homogeneous configuration. The IPC increases to 2.14 on the $2F2S$ heterogeneous configuration! Comparing the IPCs, the speedup on the 4-core heterogeneous configuration appears to be around 10% even though the latter configuration has replaced two of the fast cores with slow ones. Comparing the execution time of the benchmark on each configuration, the speedup is proved to be around -9% (slowdown). In this case, IPC would have predicted a 10% speedup, a faithful full time simulation or *TBS* would have shown close to 10% slowdown. In some other cases, the IPC increase is less than the observed speedup.

Equally important, *IBS* fails to simulate a representative overlap of the thread's execution. For example, the amount of invalidations on the caches increases by 92% on the 4-core heterogeneous configuration, while *IBS* shows around 20% increase. Also it is not clear how different threads contribute in the critical path of the benchmark execution.

Nonetheless, we show the IPC results for each application category. This is to show that IPC estimated by both sampling methods is accurate across all the application categories and configuration, while it is not a fair metric to be used for evaluation. To avoid clutter, we only show the box plot for each application category. Figure 8, shows that both *IBS* and *TBS* estimate the IPC within 5% of the full simulation on average.

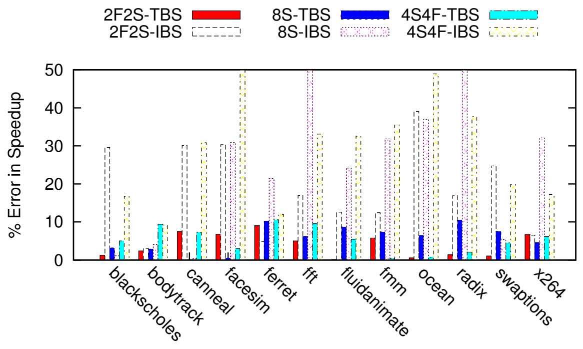


Figure 9: Breakdown of speedup for each multithreaded application. The average error is 4.99% for *TBS*.

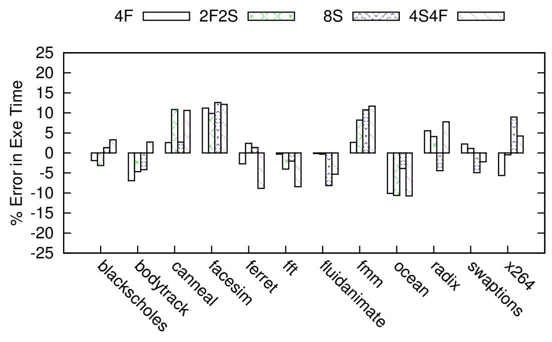


Figure 10: Absolute error in execution time estimation .

For more detailed information, Figure 9 presents the breakdown of speedup errors for multithreaded applications. There are 3 configurations ($4S$, $8S$, $4S4F$) compared against the baseline configuration ($4F$). The results are shown for both *TBS* and *IBS*. Therefore, there are 6 bars for each application to show the error in each configuration for each sampling method.

For some applications such as *bodytrack*, both *IBS* and *TBS* estimate an accurate speedup. The reason is that *bodytrack* implements an abundance of synchronizations that maintains the relative progress of the threads even in *IBS*. The sleep-based synchronization also limits the number of synchronization instructions in user mode execution. As a result, IPC correlates well with the execution time. However, *IBS* fails to consistently provide accurate estimation of speed up for all the applications due to the lack of progressed time information, and non-representative overlap of the threads.

The accuracy reported by *TBS* is consistently within 11% of the speedup reported by full simulation. Figure 10 shows the absolute error in the estimation of execution time for each benchmark across different configuration. For example, *fmm* has the highest error, which is partially due to its small execution time. As discussed in [31, 28], longer sampling units can be gathered to achieve higher accuracy based on the observed variability in the samples. While Figure 6 shows the trend in the coefficient of variation in IPC, we do not report confidence intervals for the progressed time.

5.2. Power and Temperature

Wunderlich *et al.* [31] report EPI in evaluation of single core configuration. Like IPC, EPI does not provide an accurate metric to compare multicore configurations in *IBS*. *TBS*, on the other hand, maintains the progressed time, and provides the necessary means to compare different multicore configurations for their power and energy efficiency.

While dynamic power depends on the utilization of resources, leakage is temperature dependent. Hence we report the estimation error for both dynamic and leakage power. The temperature of the processor is also evaluated for both maximum and transient values.

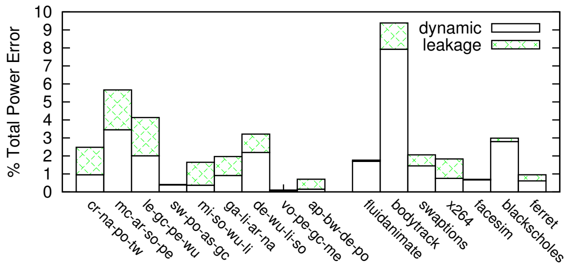
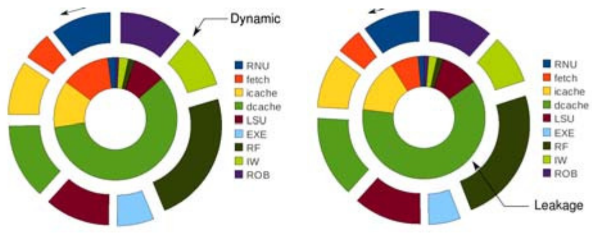


Figure 11: Error in estimated dynamic and leakage power.

Figure 11 shows the error in estimated power on average for the experiments. It also shows the breakdown of error for dynamic and leakage power consumption. Note that even though the error in dynamic power can be canceled out by the error in the leakage estimation, we compute each error separately and add them together. Application categories (*i.e.*, multiprogram vs. multithreaded) are grouped separately. Nevertheless, there is no meaningful difference between them in terms of accuracy of the estimated power. *bodytrack* has the highest error percentage among the benchmarks. The main reason is that this application has

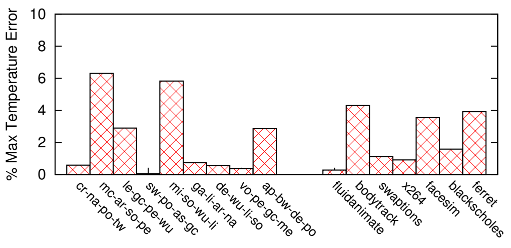


(a) Full (b) TBS

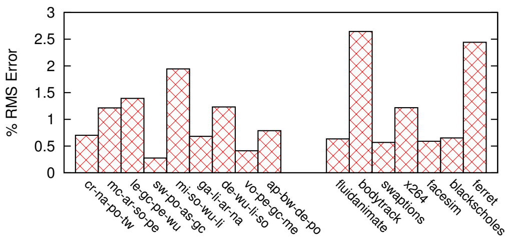
Figure 12: Breakdown of dynamic (outer ring) and leakage (inner ring) power consumption is preserved by *TBS*.

a much lower power consumption range (around a half of the *fluidanimate*), and as a result the reported percentage in higher. The average error is less then 1 Watt.

The results in Figure 12 show that the sampling preserves the breakdown of power consumption for each functional block. Figure 12a shows the breakdown for full simulation on the 4*F* configuration. The outer donut chart shows the dynamic power breakdown for 9 functional blocks in the *fast* cores (See Table 5). The inner ring shows the leakage breakdown. Figure 12b shows the breakdown for the sampled simulation. The results are gathered by averaging the power consumption of all the benchmarks.



(a) Max Temperature Error



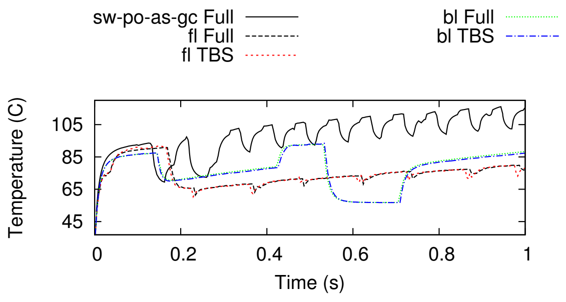
(b) RMS Error

Figure 13: Percentage of maximum temperature error in °C range (a), and the RMS error (b).

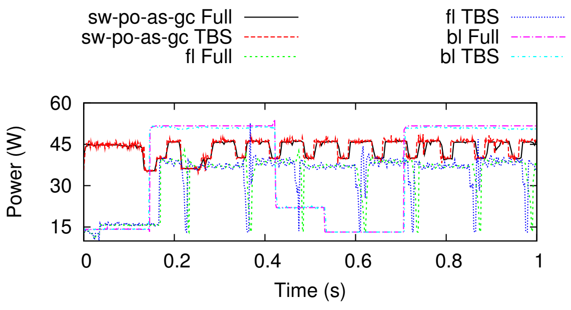
We evaluate the accuracy of temperature estimation by two metrics; Maximum temperature, and RMS error. The average temperature error across the chip for all the configurations falls under 1% compared to the full simulation. However, the temperature of the hottest blocks are critical to performance and reliability. Figure 13a shows the error in estimation of maximum temperature. The sampling results in accurate maximum temperature for both multiprogram and multithreaded applications. The absolute value of the error is within 4°C.

$$RMSE = \frac{\sqrt[2]{\frac{\sum_{i=1}^{n} Area_i \times [\sum_{j=1}^{m} (TempFull_{i,j} - TempSmpl_{i,j})^2]}{m \times totArea}}}{Temp_{max} - Temp_{min}} \quad (2)$$

In addition to average results, we also report the RMS error for temperature traces. RMS error quantifies how accurately the sampled temperature trace follows the trace of the full simulation. Equation 2 formulates how we compute RMS error. We match point to point and compute the error for each block during the execution, and scale the error by area of each block. Functional blocks are indexed from 1 to n, and the temperature samples from 1 to m. The result is normalized by the dynamic range of temperature in each application ($Temp_{max} - Temp_{min}$). Figure 13b shows the results. For example, the figure indicates that the temperature trace of the sampled simulation for *bodytrack* follows the full simulation trace with 2.64% error through the execution.



(a) Parsec temperature Trace



(b) Power

Figure 14: Temperature and power traces for some selected applications.

To gain a visual understanding of the temperature and power traces, Figure 14 shows the temperature and power traces for 6 applications during 1 second of execution. Figure 14a shows the temperature trace for two PARSEC benchmarks, *blackscholes* and *fluidanimate* along with multiprogram execution of *swim*, *povray*, *astar* and *gcc*. Temperature traces are captured from around the center of the die (happen to be a *ROB* block). Figure 14b shows traces for the total power for these applications .

5.3. Simulation Speed

The simulation infrastructure in our experiments is composed of 4 stages summarized in Table 9. QEMU user mode executes multithreaded applications with many threads, but it serializes the execution with a central spinlock. While this limits the speedup, native or distributed emulation could help the scalability of the simulation. Nevertheless, this is orthogonal to the sampling, and is not the focus of this paper.

The dynamic streams of instructions for threads are passed onto a multithreaded sampling stage. The instruction stream is cracked into micro operations for the timing model in parallel for each thread. Also the memory warmup is performed in a multithreaded way. Those instructions selected for detailed timing simulation are passed to a cycle-accurate timing model. The timing model for the whole system is performed within a single thread. The sampling is basically intended to decrease the load on this stage. If the thermal simulation is enabled, a single threaded thermal solver will perform the computation to estimate the temperature across the chip. The execution time of the thermal model depends on the size of the chip, cooling and package specification, as well as the granularity. Table 9 summarizes each stage and its functionality.

Stage	MultiThreaded	Description
S1	No	QEMU running ARM.
S2	Yes	Instruction Crack, Sampler, Memory system Warmup.
S3	No	Cycle-accurate Timing model
S4	No	Thermal computation

Table 9: Different stages in the simulation infrastructure.

The average simulation speed for the configurations evaluated in this work is over 9 million simulated instructions per second (MIPS) [4]. The thermal simulation takes around 25% of the simulation time. Consequently, the speed with thermal simulation reaches 6.5 MIPS on average. This is an order of magnitude faster than full simulation, or even distributed timing simulation. The emulation itself on average reaches the speed of 90 MIPS, while the timing simulation is around 500 KIPS on average.

5.4. Parallelization and Future Work

A distributed emulation and timing simulation such as the one performed in [18] or [7] could help scalability and increase the simulation speed. Also faster thermal simulation, for example GPGPU accelerated thermal computation, could increase the simulation speed. Nonetheless, the speedup and scalability provided by these methods is orthogonal to the speedup provided by sampling.

[4]Experiments ran on AMD Opteron(tm) Processor 6172

Performing full timing simulation even in a parallel way has limitations in speeding up the simulation. For example the average simulation speed for Sniper [7] is around 1 MIPS for an 8-core simulated configuration. This is much less than the 9 MIPS speed achieved by *ESESC*. The benefit of those techniques though is in their scalability.

To understand the scalability of available simulation frameworks, we run different available state of the art simulators. Marss [20] and SESC [21] implement timing simulation in a single thread. Sniper [7] and SlackSim [8] implement a multithreaded timing simulation. We deliberately ignore the fact that these simulators model the core with different levels of detail (For example, Sniper models the cores in higher level of abstraction, while Slacksim and *ESESC* model the core in more detail). We run each simulator with a different number of simulated cores on an AMD Opteron(tm) Processor 6172 (48 cores) with 128 GB memory. For those that we cannot run, we approximate the speed by scaling the reported speedup for the simulator based on the SPEC results [1] published for the host machine they have used for the experiments ($MIPS = reportedMIPS \times \frac{SPECRes(Opteron6172)}{SPECRes(reportedHost)}$).

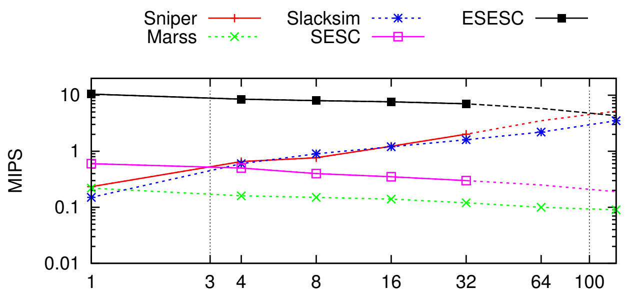


Figure 15: Simulation speed for different simulators.

Figure 15 shows the results comparing single threaded without sampling simulators (Marss, SESC) and multithreaded simulators (Slacksim and Snipper), against *ESESC*. Single threaded simulators and *ESESC* slightly decrease performance as more cores are simulated. The reason is lower cache locality and more messages in the memory hierarchy. Multithreaded simulators have nearly linear speedup for highly scalable applications. As a result, even fast simulators like SESC become slower than multithreaded when more than 3 cores are modeled.

ESESC is much faster because in the S1 stage QEMU JIT allows emulation acceleration, reaching 100 MIPS in many applications. The result is a over 9 MIPS simulation speed. Extrapolating the trend for scalability of each thread indicates that for configurations with less than 100-128 cores, *ESESC*, provides faster simulation. Note that for the extrapolation, we assume that the number of cores on the host machine scales as well, *i.e.*, a 64-core target system is being simulated on a 64-core host machine.

This indicates that *ESESC* provides faster simulation speed, up to high numbers of simulated cores even with a single threaded timing simulation. Nevertheless, sampling could be used with a multithreaded execution of the timing simulation as well, potentially delivering even higher simulation speed. We do not evaluate this idea in this paper, and it is part of future work.

6. Conclusions

This work presents Time-Based Sampling (*TBS*) framework to address challenges of sampled simulation of multithreaded applications. *TBS* reconstructs the progressed time in a sampled simulation, and specifies the sampling parameters over time instead of the instruction count. This ensures the same amount of relative progress among threads, similar to what would occur in a full simulation, which maintains the overlap of the threads. Also it enables the comparison between different architectures running multithreaded benchmarks using the execution time metric. Evaluations on a range of multithreaded applications shows that *TBS* provides an accurate estimation of the execution time while comparing different processors with varying core count, architecture, and heterogeneity. The average error across the configurations and applications is within 4.99% compared to full simulation. In addition, *TBS* is the first to enable power and temperature evaluation in a statistically sampled simulation of multicore configurations. This is done by reconstruction of power traces in addition to the progressed time. The evaluation shows an average error of 5.5% and 2.4% for power and maximum temperature respectively. Our open source simulator based on *TBS*, called *ESESC*, reaches to up to 9 MIPS of simulation speed.

7. Acknowledgments

We like to thank the anonymous reviewers, and Lieven Eeckhout for their constructive feedback. We also thank Elnaz Ebrahimi, Gabriel Southern, and Jason Duong for their help implementing the simulation infrastructure. This work was supported in part by the National Science Foundation under grants 1059442 and 1205377; Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Standard performance evaluation corporation.

[2] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society.

[3] A. R. Alameldeen and D. A. Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, Jul. 2006.

[4] E. K. Ardestani, E. Ebrahimi, G. Southern, and J. Renau. Thermal-aware Sampling in Architectural Simulation. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 33–38, New York, NY, USA, 2012. ACM.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[8] J. Chen, M. Annavaram, and M. Dubois. Slacksim: a platform for parallel simulations of cmps on cmps. *SIGARCH Comput. Archit. News*, 37(2):20–29, Jul. 2009.

[9] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi. Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):1–32, 2009.

[10] A. Coskun, R. Strong, D. Tullsen, and T. Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proceeding of the 11th International joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 169–180, Jun. 2009.

[11] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation. *The Computer Journal*, 48(4):451–459, 2005.

[12] M. Ekman and P. Stenstrom. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *In Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 89–99, 2005.

[13] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[14] J.L. Henning. SPEC CPU2000: Measuring Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[15] S. Li, A. J. Ho, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009.

[16] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *International Symposium on Performance Analysis of Systems and Software*, pages 164–171, Tucson, Arizona, Nov. 2001.

[17] F. J. M.-Martinez, E. K.Ardestani, and J. Renau. Characterizing processor thermal behavior. In *Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 193–204, 2010.

[18] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.

[19] J. N.-Battilana and J. Renau. Soi, interconnect, package, and mainboard thermal characterization. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design (ISLPED)*, pages 327–330, 2009.

[20] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[21] J. Renau, F. Basilio, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, Oct. 2002.

[23] A. Snavely and D. Tullsen. Symbiotic Job Scheduling for a Simultaneous MultithreadingProcessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Cambridge, Massachusetts, Nov. 2000.

[24] N. Tuck and D. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 26–34. IEEE, 2003.

[25] M. Van Biesbrouck, T. Sherwood, and B. Calder. A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. In *International Symposium on Performance Analysis of Systems and Software*, pages 45–56, Austin, Texas, Mar. 2004.

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

[27] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Turbosmarts: accurate microarchitecture simulation sampling in minutes. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 408–409, New York, NY, USA, 2005. ACM.

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

[29] T. F. Wenisch and R. E. Wunderlicha. SimFlex: Fast, Accurate and Flexible Simulation of Computer Systems. *Tutorial in the International Symposium on Microarchitecture MICRO-38*, 2005.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[31] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. 30th Annual Int Computer Architecture Symp*, pages 84–95, 2003.