Transparent Sampling

Taj Muhammad Khan
INRIA Saclay, France
taj.khan@inria.fr

Daniel Gracia Pérez
CEA, LIST, Software Safety Laboratory,
91191 Gif-sur-Yvette CEDEX, France
daniel.gracia-perez@cea.fr

Olivier Temam
INRIA Saclay, France
olivier.temam@inria.fr

***Abstract*—Low simulation speeds have a critical impact on the design process by limiting the number of design options which can be explored. Sampling is a popular fast simulation technique because it can achieve high simulation speed and high accuracy. However state-of-the-art sampling techniques either consider warm-up as an orthogonal issue and leave the choice of a warm-up technique to the end user, or require cumbersome simulator modifications from the end user. Since the most user-friendly and efficient warm-up techniques are not easily compatible with the most efficient sampling techniques, the end user is left with a difficult choice, or runs the risk of misusing sampling techniques with poor warm-up. *Transparent Sampling* reconciles sampling and warm-up techniques by delivering state-of-the-art accuracy and simulation time, while remaining easily accessible to end users not proficient in, or not willing to delve into, fast simulation issues.**

I. Introduction

There are multiple approaches for tackling the simulation speed issue, such as abstract simulation [11], parallel simulation [15], statistical simulation [21], synthetic benchmarks [13], [22], and sampling [17], [21]. Among these, sampling, i.e., functional simulation of all the program except for a few samples which are simulated in detail (performance simulation), has demonstrated the capability of improving simulation speed by several orders of magnitude with little loss in simulation accuracy.

However, in spite of their popularity, sampling techniques usually require an effort from (and are thus not transparent to) the user for at least one of three reasons: warm-up, simulator-level modifications or frequent target program modifications (e.g., co-design). Let us consider the two state-of-the-art sampling techniques in that context, SimPoint [17] and SMARTS [21].

SimPoint is a phase-based sampling strategy: all trace intervals are clustered into phases, and the samples are representatives of each phase. SimPoint is also called *representative sampling*. SimPoint requires no simulator-level modification. However, it considers warm-up to be an orthogonal issue and leaves the choice of the warm-up technique to the user. Still, two state-of-the-art warm-up techniques, MRRL [9] and BLRL [4], are compatible with Simpoint. However, the original SimPoint technique is cumbersome in the presence of frequent target program modifications: the user must run a functional simulation pass after each modification in order to regenerate the Basic Block Vectors (BBVs) used to select the samples; sampling itself (and performance measurement) is performed within a second pass. The authors of SimPoint later addressed that issue with on-line SimPoint [16] which gets rid of the two-pass requirement, but at the expense of dynamically selecting the sampling intervals during simulation. However, dynamically selecting sampling intervals makes it impossible to use MRRL or BLRL, because these techniques also require a functional pass to analyze the trace before each sample. The only state-of-the-art warm-up technique which does not require two passes is SMA [14]: it monitors the warm-up status of each SRAM during simulation and authorizes sampling only when they are all deemed warmed. However, as a result, SMA shifts by an undetermined number of instructions the beginning of the sampling interval. This, in turn, does not make it compatible with on-line sampling either, which relies on a phase analysis strategy for predicting which sample should be collected next. So, even though it is a one-pass approach, it precisely sets the beginning of each sample.

The issues with SMARTS are different. SMARTS is a *statistical sampling* approach: samples are selected regularly (or they could also be selected randomly) within the trace. SMARTS is a single-pass sampling strategy so it is compatible with frequent target program modifications. SMARTS also relies on continuous warming so no other warm-up technique is necessary. However warming up is implemented within the functional simulator: the user has to replicate all SRAM structures to be warmed up in the functional simulator, at the cost of a (possibly significant) software engineering effort. Moreover, for time-sensitive mechanisms, such as prefetching (e.g., to warm-up prefetch buffers), it can be difficult to implement the warm-up altogether since the functional simulator has no notion of time.

As a result, while both techniques are remarkably accurate and fast, they are not fully *transparent*: the user either has to put up with the overhead of two-pass sampling in case of frequent target program modifications (original SimPoint), or has to use non-state-of-the-art warm-up strategies, e.g., fixed-size warm-up (on-line SimPoint), or has to endure functional simulator modifications for warm-up with a sometimes imprecise result (SMARTS).

In this article, we propose a sampling strategy

978-1-4244-7938-2/10/$26.00 ©2010 IEEE

that achieves transparency on all three counts: seamless warm-up adaptation to the size and number of SRAM structures, almost no simulator modification and single-pass sampling for accommodating frequent target program modifications. At the same time, this strategy achieves state-of-the-art accuracy and simulation speed (fraction of all instructions for which detailed/performance simulation is required). Moreover, like SMARTS [21], the technique provides an estimate of the confidence interval. Finally, the technique has been designed so that the user need not set any internal parameters whatever the number and size of SRAM structures: the only parameters exposed to the user are the desired fraction of instructions to be performance simulated, and the desired confidence interval.

II. Principles

We now consider which warm-up and sampling approaches are compatible together. As noted before, online representative sampling is not compatible with either static or adaptive warm-up. For that reason, Transparent Sampling (TS) is based on statistical sampling, with samples randomly selected. However, we do not want to resort to continuous warm-up as in SMARTS in order to avoid functional simulator-level modifications; we want to warm up SRAM structures just using the performance simulator. Therefore, we turn to adaptive warm-up, such as SMA[14]. The difficulty is then to design a joint statistical sampling + adaptive warm-up technique which will deliver good performance, i.e., high accuracy and small fraction of total instructions performance simulated, and still remain entirely transparent for the user, i.e., no parameters to set except ones easily comprehensible by the user.

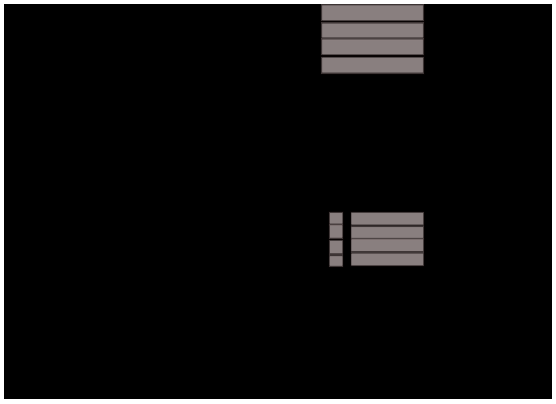

Fig. 1. *SRAM library.*

A. Warm-Up

In order to achieve transparent sampling, we warm up using the performance simulator: by simply running, the performance simulator will progressively warm up the SRAM structures. The simulation is partitioned into fixed-size intervals of N instructions. At any interval, we assess the degree of warm-up. For that purpose, we monitor the load/store references that access SRAM entries which are already warmed, as in SMA [14]. In order to transparently implement this monitoring we provide a simple SRAM library (in C++ or C) on top of which any cache or table can be easily implemented. This is the only simulator modification required to use Transparent Sampling, and it is benign: the user only needs to replace the classic array or object declaration used for SRAM structures with an SRAM class instantiation (or a call to a function in C), see Figure 1. This SRAM class implements a single bit per SRAM entry. This bit is reset by the Transparent Sampling Engine (TSE) before each sample, and set at the first reference within a sample. Thanks to this bit, it is possible to determine whether each access (e.g., load/store, branch prediction table access, etc) is cold or warm without any further simulator modification by the user.

So far, this warm-up technique is very similar to SMA [14], except that we provide the library and class support. Unlike SMA, we do not use the fraction of SRAM structures which are warmed as a warm-up criterion, because we empirically observed this criterion to be highly sensitive to the program behavior. For instance, some program parts with a small workload will only warm up a fraction of the cache. SMA combines both warm-up criteria but we found that solely using the fraction of warm accesses was more robust. We deem that all the SRAM structures are warmed when the fraction of warm accesses to each of these SRAM structures (e.g., the different caches of a memory hierarcy) is above a threshold.

However, even though our adaptive warm-up technique makes use of the aforementioned warm-up criterion, it does not rely on the variable-length warm-up interval of SMA for the following reason. We found that SMA can have a non-trivial impact on the *selection of intervals used for performance measurement*, the interval immediately following the warm-up intervals; more exactly, that it could shift the performance measurement intervals in a way that could be degrading the randomness of performance measurements (a key aspect of statistical sampling). Because SMA was studied as a warm-up technique alone (as opposed to a warm-up + sampling technique), it is only normal that this effect has gone unnoticed, but we found it to be severely detrimental in some cases. Consider a program where the following pattern recurs often: a region with accesses to many distinct addresses followed by one or several region(s) with repeated accesses to a few addresses. The first region is likely to breed significantly more cold accesses than the second region(s). As a result, the fraction of cold accesses in the first region will be high, and the warm-up threshold won't be passed. When the program enters the second region(s), the fraction of warm accesses quickly increases because just a few

addresses are being repeatedly used, and the threshold is likely to be passed. As a result, adaptive warm-up has shifted the performance measurement interval to a region with few repeated accesses. Now, the first region is likely to exhibit a higher *miss rate* than the second region because a higher number of distinct addresses are accessed. As a result, adaptive warm-up has *shifted the performance measurement interval to a region with a lower miss rate*.

Note that, in a more complex case where a many-address region follows a few-address region which follows again a many-address region, adaptive warm-up could have the exact opposite effect and shift the performance measurement interval to a many-address region, artificially increasing the measured miss rate. We observed both cases. Note that these cases are not frequent, and adaptive warm-up using only the threshold of warm accesses often works well; but in some cases, this bias severely degrades the accuracy of sampling, making the technique less robust.

So we need to avoid the shifting effect of adaptive warm-up, but at the same time, we do need adaptive warm-up in order to adapt to variable SRAM sizes. In order to reconcile both constraints, we proceed as follows. At each sample, we measure the warm-up size necessary to pass the threshold, and using all such measurements since the beginning of the execution, at any sample, we compute the *average warm-up size*. At the next sample, we use this average warm-up size as the warm-up length, independently on the threshold. After a few samples, the average warm-up size stabilizes, and this is almost akin to a fixed-size warm-up. As a result, performance measurement intervals are almost shifted by a constant number of instructions, avoiding to bias the randomness of their selection. Even though we do not factor in the threshold for stopping the warm-up, we monitor it. In case the warm-up threshold has still not been reached after the performance measurement interval, *we let the performance simulation carry on until the threshold is reached*, see Figure 2, but we do not use the corresponding intervals for performance measurement nor warm-up; they are simply used to allow us to compute the new average warm-up size.

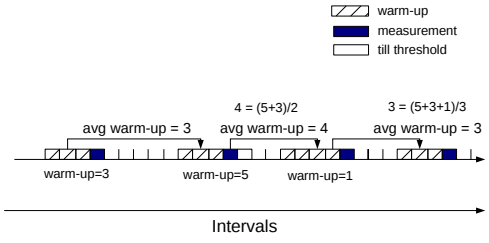

Fig. 2. *Computing the average warm-up size.*

We found this approach to be robust and to bring the benefits of both worlds: the almost constant warm-up size avoids biasing the selection of performance measurement intervals, but the warm up size does depend on (automatically adapts to) the size of SRAM structures.

B. Rolling window

In theory, the aforementioned warm-up threshold, used to decide when the SRAM structures are warm, is potentially architecture-dependent. Thus it should normally be exposed to the user; however, we implemented a safeguard which allows using a fixed threshold whatever the size and number of SRAM structures.

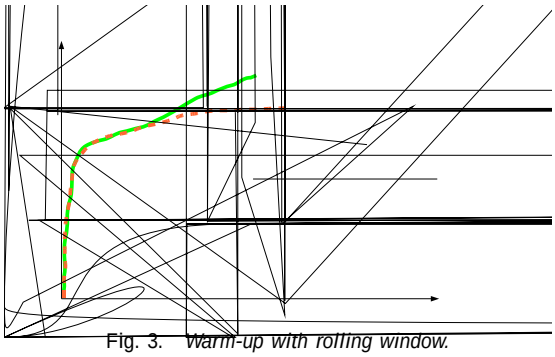

Fig. 3. *Warm-up with rolling window.*

The safeguard is that the fraction of warmed accesses is not computed based on all intervals since warm-up started but based on a rolling window of intervals. In Figure 3, we see that, when the number of performance simulated intervals is equal to the size of the rolling window, i.e. the window starts rolling leaving the initial intervals behind, the fraction of warmed accesses based on the rolling window intervals starts to increase faster than the fraction based on all intervals since the beginning of the sample. In other words, the large fraction of cold accesses in the first few intervals has no impact after the rolling window has shifted, and ultimately the fraction of warm accesses in the rolling window converges more quickly to 100%. While both simulation time and accuracy are good with a threshold set at 100%, we empirically found that we could achieve the same accuracy with fewer simulated instructions with a fixed threshold of 99.9% and a rolling window of 100 intervals across all tested programs and architectures. The absence of rolling window results in longer warm-ups. This breeds either more simulated instrucitions per sample, and therefore more overall simulated instructions, or, if we try to keep the percentage simulation constant, fewer samples.

C. Sampling objectives

The only role of the user is to set sampling goals. Essentially, the user has to set a simulation accuracy/time toggle. For that purpose, only two criteria are exposed to the user: the fraction of total instructions which are performance simulated (the number of instructions in

samples over the total number of instructions executed), and the confidence interval of the results. The user can set either one, or both criteria. These criteria will drive the simulation. The outcome of the simulation is the confidence interval of the CPI, using simple statistical techniques [21], and the fraction of instructions which are performance simulated.

1) Objective: fraction of performance simulated instructions: The aforementioned criteria are used to drive the sample selection. For now, we will only consider the fraction of total instructions to be simulated. The program is run using the functional simulator. The simulation is partitioned in intervals of size N. At any interval, the TSE must decide if the next interval will be performance simulated (sample collected) or just functionally simulated. For that purpose, it monitors the fraction of instructions performance simulated so far F_S, and the average warm-up size W_S (in number of intervals). Between two samples, as instructions are only functionally simulated, F_S decreases; when performance simulating instructions for a sample (warm-up and measurement), F_S increases.

Let F be the fraction of performance simulated instructions that the user requested. F can be interpreted as the probability that one interval should be performance simulated. However, when sampling is triggered, on average $W_S + 1$ intervals are performance simulated: W_S for warm-up + 1 for measurement. So, assuming a uniform distribution of samples, the probability that a sample (warm-up + measurement) is collected is $\frac{F}{(W_S+1)}$.

F is actually the *initial* probability that a sample is triggered at the next interval. As simulation progresses, the actual fraction of performance simulated instructions F_S will oscillate around F. If F_S exceeds F, the number of samples should be reduced, or conversely can be increased if F_S is less than F. Therefore, we use $\frac{F}{(W_S+1)}$ as the probability to sample at the next interval. W_S is adjusted as a function of F_S and hence controls the simulation probability. Since the user-specified bound is statiscally enforced, the resulting number of performance simulated instructions will not exactly match the bound, but we found that this criterion allows it to fall reasonably close in all cases.

Note that this sampling selection criterion is robust in spite of adaptive warm-up. If the SRAM structures are large, W_S will increase, as a result $\frac{F}{(W_S+1)}$ will decrease resulting in fewer samples. Conversely, if the warm-up requirements are low, the number of samples will be increased which will have a positive effect on accuracy, while remaining within the simulation time bounds set by the user.

2) Objective: confidence interval: A user may want to focus on simulation speed or accuracy. The above criterion focuses on speed by constraining the fraction of performance simulated instructions. It is also possible to focus on accuracy by constraining the confidence interval size. After simulation, an approximation of the confidence interval can be derived using the standard deviation of the target metric (CPI in our experiments).

The Central Limit Theorem states [8] that if we have sufficient number of independant observations from the same population which has a mean μ and standard deviation σ, the sample mean follows a normal distribution with mean μ and standard deviation $\sigma/\sqrt{n}$. The confidence interval of the normal distribution is then determined by $(\overline{x} - [c * s/\sqrt{n}], \overline{x} + [c * s/\sqrt{n}])$, where n is the number of observations, $\overline{x}$ is the mean of a measurement X, s is its standard deviation and c is derived from the percentile of the distribution satisfying our confidence requirement. Because the actual mean of a measurement cannot be obtained unless all the population is known (in the case of the CPI, it would mean performance simulating all the program intervals), only an approximate mean, corresponding to the mean of the sampled population, is used. As a consequence, we can only provide an *estimate* of the confidence interval. However, we found 99.7% estimate to be satisfying in most cases.

The confidence interval can be used to guide the selection of the percentage of performance simulated instructions. If the confidence interval size does not fulfill the user's accuracy objectives, the user can progressively increase the fraction of simulated instructions until this objective is reached.

III. Methodology

Processor	Simplescalar, out-of-order, 4 way
RUU/LSQ Size	16/8
Pipeline	5 stages
D-Cache/I-Cache	64KB, 2-way 32-byte block
Cache/Memory Latency	2/60 cycles
Memory Ports	2
Registers	32 Int, 32 FP
Functional Units	4 I-ALU, 1 I-Mult 4 F-ALU, 1 F-Mult 4 F-ALU, 1 F-Mult
I-TLB	16 4KB 4-way assoc blocks:lru
D-TLB	32 4KB 4-way assoc blocks:lru
Branch miss penaly	3 cycles miss lat

TABLE I
Simulator configuration.

We used the SimpleScalar [2] 4-way out-of-order processor using the Alpha ISA, see Table I. We modified the architectural structures to render it compatible with the TSE, by calling our SRAM library (which implements one bit for each line of the cache structures).

We used the SPEC2000 [7] benchmark suite to evaluate our sampling technique. Both SPECINT and SPECFP programs were used with *ref* input sets. In order to demonstrate the resilience of our technique to architectural changes, especially SRAM structures sizes, we

Program	# intervals	missrate 8k	missrate 64k	missrate 128k
bzip	1.08878e+07	0.0339	0.0214	0.0188
crafty	1.91883e+07	0.0922	0.007	0.003
eon	8.06141e+06	0.03	0.0008	0.000
gap	2.69036e+07	0.0276	0.0165	0.0155
gcc.166	4.69177e+06	0.0829	0.073	0.0721
gcc.200	1.08626e+07	0.0622	0.04	0.0329
gzip	8.43674e+06	0.1118	0.0307	0.0091
mcf	6.18675e+06	0.4109	0.3927	0.3845
perlbmk	3.99293e+06	0.0322	0.0058	0.002
twolf	3.46485e+07	0.1082	0.0793	0.0701
vortex	1.18972e+07	0.0263	0.0106	0.0072
vpr	8.40688e+06	0.0783	0.0442	0.0378
ammp	3.26549e+07	0.0898	0.0641	0.0535
applu	2.23884e+07	0.139	0.1106	0.109
apsi	3.47923e+07	0.0834	0.044	0.0405
art	4.17954e+06	0.4024	0.4007	0.3993
equake	1.31519e+07	0.1837	0.1529	0.1484
facerec	2.11027e+07	0.0462	0.0377	0.0376
fma3d	2.68368e+07	0.073	0.0414	0.0408
galgel	4.09355e+07	0.1631	0.057	0.0549
lucas	1.42399e+07	0.1661	0.165	0.1649
mesa	2.81691e+07	0.0197	0.0047	0.0036
mgrid	4.19156e+07	0.1691	0.0679	0.0667
sixtrack	4.70949e+07	0.0142	0.0041	0.0041
swim	2.25831e+07	0.173	0.1729	0.1585
wupwise	3.49624e+07	0.0345	0.0278	0.0271

TABLE II
Benchmarks characteristics.

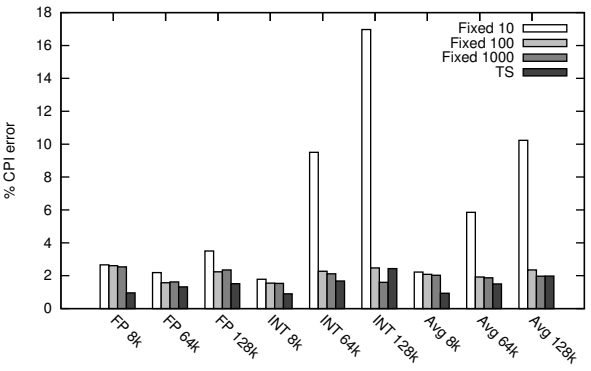

Fig. 6. *TS vs. fixed warm-up (% CPI error).*

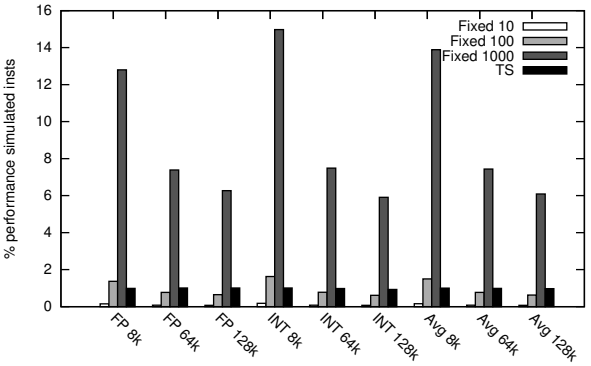

Fig. 7. *TS vs. fixed warm-up (% performance simulated instructions).*

vary the cache sizes from 8KB to 128KB. Detailed simulations of the full benchmark suite (no sampling) are used to obtain the actual CPI (Cycles Per Instruction) of the programs. The benchmarks used and some of their characteristics are listed in Table II.

IV. Performance Evaluation

We evaluate Transparent Sampling in this section. The two main metrics are accuracy and simulation time. Accuracy is defined as the CPI error of the sampled simulation versus the full simulation. Simulation time is correlated to, and thus defined as, the fraction of the total instructions in the program trace that were performance simulated (as opposed to only functionally simulated). We evaluate Transparent Sampling using the percentage of simulation objective, which we set at 1%. Since one of the key purposes of our technique is to accommodate architecture modifications, especially SRAM structures sizes modifications, all results are provided for three different cache sizes: 64KB is the baseline L1 cache size, and we also experiment with a smaller (8KB) and a larger cache size (128KB).

A. Transparent Sampling with an objective 1% of performance simulated instructions

The percentage of performance simulated instructions is indicated in Figure 5. Even though the TSE uses a statistical control mechanism, the percentage is successfully maintained below 1.12% for all programs. The CPI error is indicated in Figure 4 for all benchmarks for the three aforementioned cache sizes. We first note the low absolute average error, less than 2%, which is on par with the best sampling accuracy results [17], [21]. Moreover, this accuracy is stable across all three cache sizes, and does not degrade much with the cache size: the accuracy is 0.91% on average for 8KB, 1.47% for 64KB and 1.93% for 128KB. In other words, the accuracy does not change much as the SRAM structure size increases. To our knowledge, this is the first demonstration of a sampling strategy that exhibits stable accuracy as architecture characteristics change.

Even though the overall error is low, some programs such as

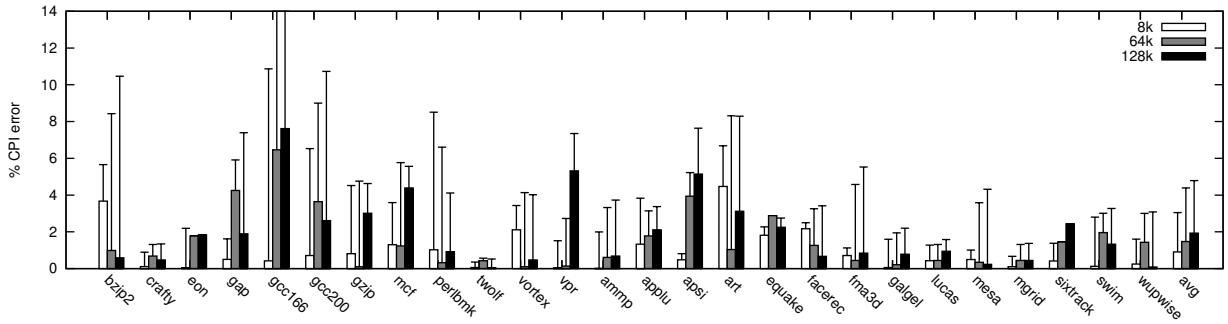

Fig. 4. *CPI error (8KB, 64KB and 128KB caches).*

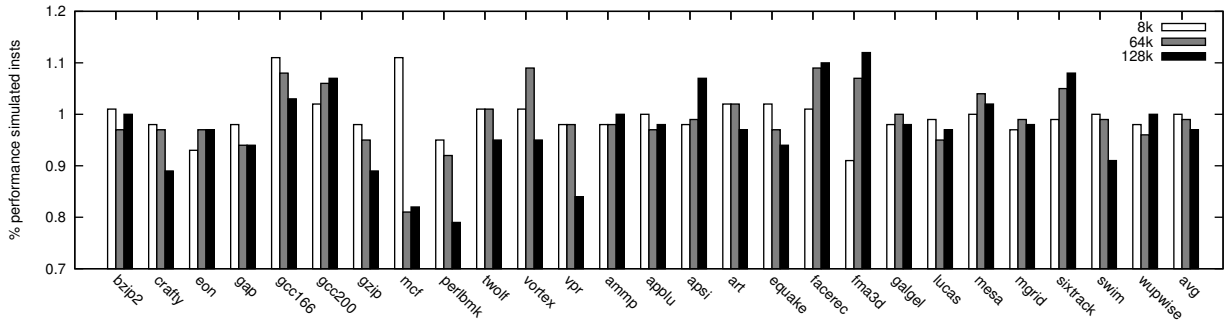

Fig. 5. *% of performance simulated instructions (8KB, 64KB and 128KB caches).*

intervals. While, for any architecture, there exists a sweet spot where the fixed warm-up size would realize a good accuracy/simulation time tradeoff, such as 100 intervals in this case, the approach is not robust. As the architecture characteristics change (e.g., larger caches), the warm-up size can become underestimated as the 10-interval warm-up results show in Figure 6: it performs well for small caches (8KB), but poorly for medium and large caches (64KB and 128KB). Conversely, the warm-up size can be overestimated as the 1000-interval warm-up results show in Figure 7.

The confidence interval indicates the interval of the estimated CPI. The upper bound of the confidence interval thus represents the estimated maximum CPI error. In Figure 4, one can notice there is no line (corresponding to the confidence interval) on top of bars for `eon`. This occurs because either the CPI is close to the upper bound of the confidence interval, or because it is even beyond. While the former case is just the maximum error, the latter case can occur because the confidence interval can only be *estimated* as explained in Section II.

B. Iteratively bounding the error

While it is likely a user would start with a low a percentage of performance simulated instructions (time) objective, the user may also want to set accuracy objectives. By taking advantage of the estimated confidence interval, a user can iteratively increase the percentage of performance simulated instructions until the desired time/accuracy tradeoff is reached. We illustrate this process below by setting a target of 5% maximum error (10% confidence interval size), and apply this iterative process to benchmarks which had confidence intervals greater than 10% of their CPI.

For each cache benchmark pair, we individually increase the percentage of performance simulated instructions to 2%, 5% and 10% until our accuracy goal is reached. In Figure 8, we show the resulting error and confidence interval when applying the iterative process; the target fraction of instructions used is indicated on top of each bar when it is different from 1%. For most benchmarks, a simulation of 2-5% reduces the confidence to the desired level, but some benchmarks require about 10% to achieve the same result, especially `gcc.166` due to its small size. `bzip2` exhibits an error of 0.6% to 4% for an increase of the number of simulated instructions from 1% to 5%, but the error still remains within the confidence interval, which is less than 5%. On average the iterative bounding of the error reduced the CPI error to less than 1.2% for the three cache sizes under study.

C. Warm-Up only based on threshold

In this section, we evaluate the impact of using a warm-up strategy solely based on the threshold of warmed accesses, which is closer to SMA. Again, SMA as a warm-up strategy performs quite well, but we here focus on its impact on sample selection. In Figure 9 and 10 we respectively report the error and the percentage of performance simulated instructions. On average, we observe that this warm-up strategy works well, with lower errors for some programs such as `bzip2` and

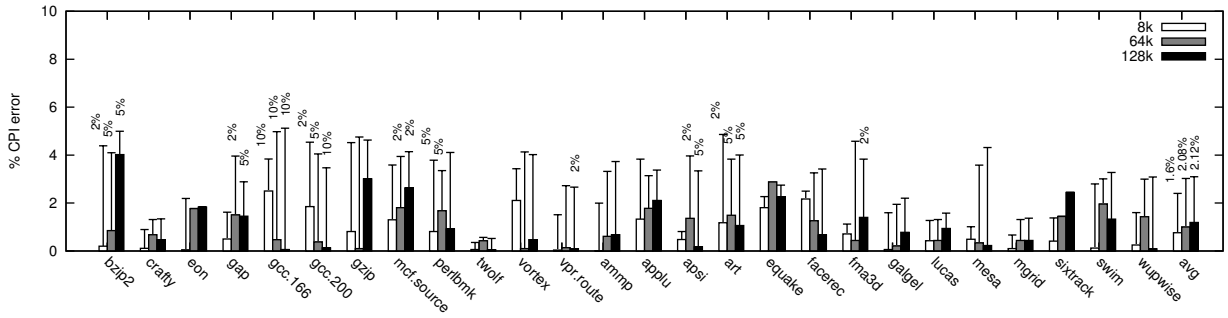

Fig. 8. *TS after iterating, with 5% error target (% CPI error).*

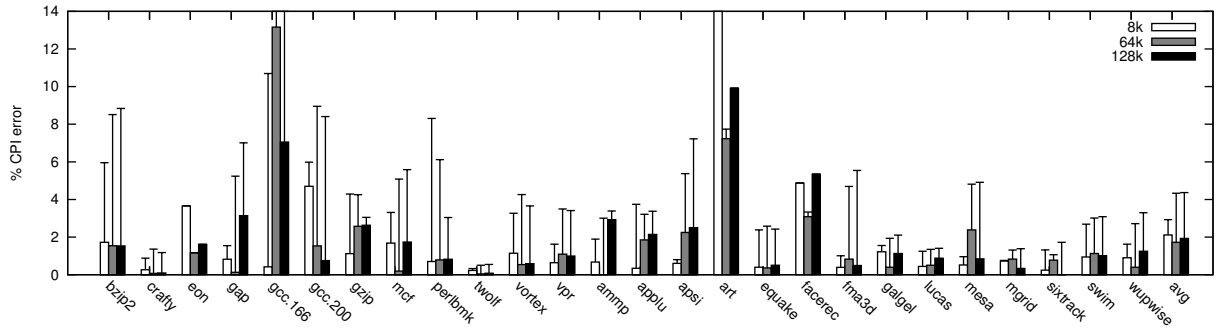

Fig. 9. *TS using only warm-up threshold (% CPI error).*

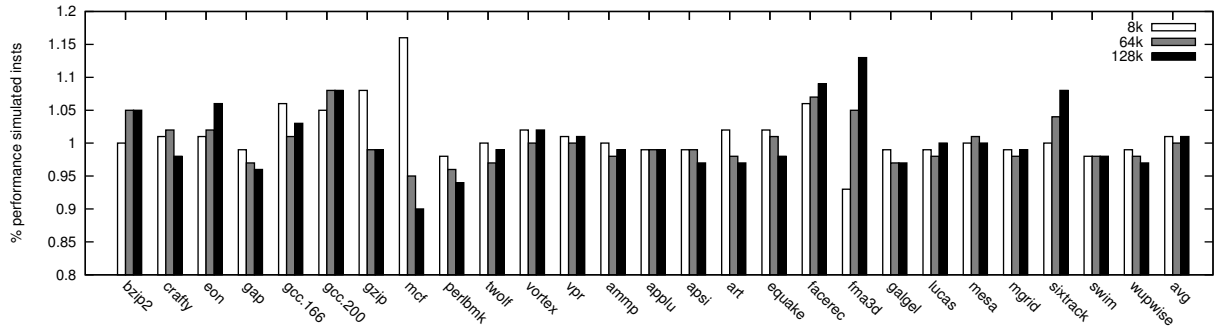

Fig. 10. *TS using only warm-up threshold (% performance simulated instructions).*

`equake`, and with smaller confidence intervals for some programs such as `twolf` and `lucas`. However, for a few programs, especially `art` and `gcc.166`, the error and the confidence interval become quite large. For these two programs, we observed the shifting effect described in Section II, which is successfully controled by our alternative adaptive warm-up approach.

D. Warm-Up length

The warm-up length (in number of intervals) is analyzed in more details in Figure 11. For the smallest cache size, programs require only 70 warm-up intervals on average. For the 64KB cache, most programs require a warm-up size up to 136 intervals on average. For the largest cache (128KB), while most programs accommodate a constant warm-up length of 181 intervals, about 3 programs require around 240-300 warm-up intervals on average. One particular example, `sixtrack`, requires 529 intervals for 128k. This result confirms that it is not appropriate to use a constant warm-up interval, across architecture configurations, or even across benchmarks for the same architecture configuration.

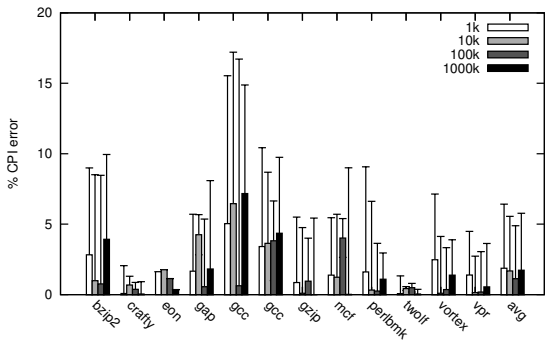

Fig. 12. *Impact of interval size (% CPI error).*

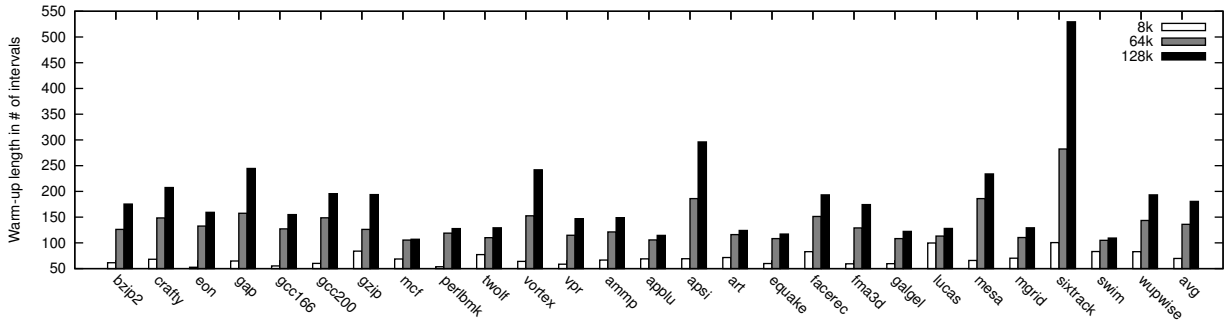

Fig. 11. *Warm-Up length as a function of cache size.*

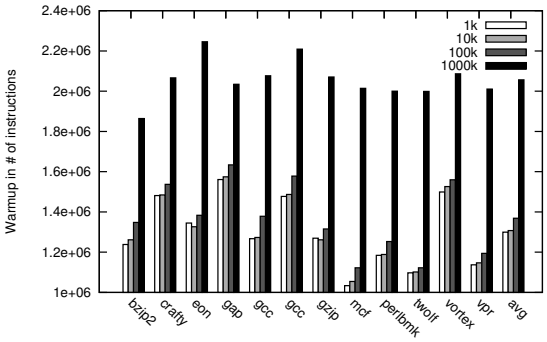

Fig. 13. *Impact of interval size (% performance simulated instructions).*

E. Interval size

Figures 12 and 13 respectively show the CPI error and the warm-up length in number of instructions when varying the interval size from 1,000 instructions to 1,000,000 instructions for 64KB caches. The counter-intuitive best choice is to use a small interval size of 10,000 or 100,000 instructions (we use 10,000 instructions throughout this study). It is counter-intuitive because the smaller the intervals, the higher the fraction of performance simulated instructions used for warm-up rather than measurement, since the warm-up size is constrained by the SRAM structures. However, small intervals enable to achieve higher accuracy by multiplying the number of samples. This tradeoff underlines that it is necessary to dedicate most of the performance simulation time to warm-up rather than measurement. Note that the 1,000-instruction intervals provide no significant benefit in simulation time at a noticeable cost in accuracy. While the results are only shown for 64KB caches and SPECINT programs due to paper length constraints, they are consistent across all cache sizes and benchmarks.

V. Related work

As mentioned in the introduction, there is a large body of work on sampling and warm-up, but much less on joint sampling/warm-up techniques. In on-line SimPoint [16], the authors propose to keep the last 50000 memory and branch events preceding the samples to warm up the states of caches and branch predictors. But the impact of this parameter on accuracy, and its dependence on SRAM structures sizes is not explored.

SMARTS [21], and similar approaches such as PGSS [12], propose to perform a continuous warm-up of the caches and branch predictors during the functional simulation. However, they require functional simulator modifications, which are cumbersome and not compatible with all architecture mechanisms, and they further slow down the functional simulators. Fast simulators relying on binary translation [10], or native execution [5], can significantly speed up the functional phase between samples, but the necessity to warm up several mechanisms can outweigh many of the benefits of fast functional simulation. In order to alleviate these issues, an alternative approach is to use checkpointing, e.g., TurboSMARTS [20], MHS [18], NSL [3], and NSL-BLRL [19]. The principle is to execute the full benchmark using a detailed simulator and to keep the states of the SRAM structures, such as caches and branch predictors, before each sample. While these techniques drastically improve simulation speed over functional simulation, they have several shortcomings for typical usage scenarios. First, the state recorded in the checkpoint is for a given architecture configuration. In order to cover some architecture variations, TurboSMARTS proposes to keep the status of a big cache and branch predictor, and apply conversion techniques to reuse the checkpoints for smaller caches and branch predictors. However, this approach still cannot capture the state of novel architecture mechanisms, or even novel cache mechanisms. Second, checkpointing techniques are not tolerant to target programs modifications: for any such modification, all the checkpoints must be regenerated.

The same goes for NSL-BLRL: instead of recording micro-architecture states, they study events which can affect micro-architecture states (e.g., load/stores, branches) and determine the number of instructions which must be played, before the sample, in order to reach a warmed state. Again, they may not capture the events appropriate for novel architecture mechanisms, the size of the recording bounds the size of the target SRAM structures, and they are similarly sensitive to program modifications. *Reverse State Reconstruction* [1]

operates slightly differently, by reconstructing the states just before the samples (the trace is played in reverse order until the states are stable) instead of modifying the states before the samples, but the limitations remain.

BeeRS [6] is a joint sampling/warm-up approach, where the warm-up is performed using the detailed/performance simulator, as in our case. The main idea is to efficiently distribute simulated instructions between warm-up and performance statistics collection, for each sample. Because BeeRS relies on variable-size intervals, it will allocate less warm-up to large intervals, and will allocate more instructions overall to samples which are the best representatives of their clusters. Besides the complexity of variable-size intervals, BeeRS does not self-adjust the warm-up size depending on the target SRAM structures.

VI. Conclusions and Future Work

In this article, we present *Transparent Sampling*, a sampling technique that reconciles sampling and warm-up techniques by delivering state-of-the-art accuracy and simulation time, while remaining easily accessible to end users. *Transparent Sampling* achieves a CPI error of 1.47% with 20.7 million performance simulated instructions (1%) on average for 64K caches.

We plan to extend this work in two ways. First, we are investigating plugging our sampling engine into a broadly distributed simulation framework such as SystemC. The adaptive nature of Transparent Sampling makes it well suited to a simulation engine-level implementation because the sampling approach can adapt to the target architecture, and because it is also well suited to frequent target program modifications, a common issue in embedded systems where co-design is common practice and for which SystemC is widely used.

Second, we have focused on single-core architectures for now. We plan to evaluate and adapt our sampling approach to multi-core architectures.

References

[1] P. Bryan, M. Rosier, and T. Conte. Reverse State Reconstruction for Sampled Microarchitectural Simulation. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:190–199, 2007.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

[3] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 468–477. IEEE Computer Society, 1996.

[4] 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, 5 2005.

[5] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor Performance Estimation using Hybrid Simulation. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 325–330, New York, NY, USA, 2008. ACM.

[6] D. Gracia Pérez, H. Berry, and O. Temam. Budgeted Region Sampling (BeeRS): do not separate sampling from warm-up, and then spend wisely your simulation budget. *International Symposium on Signal Processing and Information Technology*, 0:1–6, 2005.

[7] J. L. Henning. Spec cpu2000: Measuring CPU performance in the new millennium. *Computer*, 33:28–35, 2000.

[8] R. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling / Raj Jain*. Wiley, New York :, 1991.

[9] J. John W. Haskins and K. Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. *ISPASS '05: IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.

[10] D. Jones and N. P. Topham. High speed CPU simulation using LTU dynamic binary translation. In *HiPEAC*, pages 50–64, 2009.

[11] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 876–881, Washington, DC, USA, 2005. IEEE Computer Society.

[12] J. L. Kihm, S. D. Strom, and D. A. Connors. Phase-Guided Small-Sample Simulation. In *ISPASS*, pages 84–93. IEEE Computer Society, 2007.

[13] A. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, 2002.

[14] Y. Luo, L. K. John, and L. Eeckhout. Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation. In *Proceedings of the 16th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 10–17, Foz do Iguacu, PR - Brazil, 10 2004. IEEE Computer Society Press.

[15] D. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. August, and D. Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. *High-Performance Computer Architecture, International Symposium on*, 0:29–40, 2006.

[16] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 321–326, New York, NY, USA, 2005. ACM.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, 2002.

[18] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2005)*, pages 47–67, Barcelona, Spain, 11 2005. Spinger Verlag.

[19] L. Van Ertvelde, F. Hellebaut, L. Eeckhout, and K. De Bosschere. NSL-BLRL: Efficient cache warmup for sampled processor simulation. In *ANSS '06: Proceedings of the 39th annual Symposium on Simulation*, pages 168–177, Washington, DC, USA, 2006. IEEE Computer Society.

[20] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. *SIGMETRICS '05*, June 2005.

[21] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.

[22] J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. Lilja, and L. John. Evaluating Benchmark Subsetting Approaches. *IEEE Workload Characterization Symposium*, 0:93–104, 2006.