Budgeted Region Sampling (BeeRS): Do Not Separate Sampling From Warm-Up, And Then Spend Wisely Your Simulation Budget

Daniel Gracia Pérez, Hugues Berry, Olivier Temam
gracia@lri.fr,{hugues.berry,olivier.temam}@inria.fr
INRIA Futurs, France

Abstract. While the recent surge of research articles on sampling started with rather large sample sizes, it has later shifted to very small intervals, and it is now converging to intermediate sizes, and even to varying sizes. With 100M samples, warm-up is not an issue, at least with current cache sizes. However, with significantly smaller samples, warm-up becomes critical, especially when the sampling target accuracy is of the order of a few percent. However, in most sampling research works, warm-up has largely been treated as a separate issue.

In this article, we advocate for an integrated approach at (simulator-based) warm-up and sampling. Instead of separating warm-up and sampling, we take exactly the opposite approach, provide a common instruction *budget* for warm-up and sampling, and we attempt to spend it as wisely as possible on either one.

This budget and integrated approach at warm-up and sampling achieves an average CPI error of 1.68% on the 26 Spec benchmarks with an average sampling size of 288 millions instructions, and at the same time, it relieves the user from any delicate decision such as setting the sampling or warm-up sizes, thanks to the integrated warm-up+sampling and the region partitioning approaches.

1 Introduction and Related Work

While the recent surge of research articles on sampling started with rather large sample sizes (100M in the first SimPoint article [12]), it has later shifted to very small intervals (1,000 in SMARTS [14]), and it is now converging to intermediate sizes (1M and 10M in SimPoint [10, 6]), and even to varying sizes in EXPERT [7] and SimPoint VLI [5] (ranging from 52,000 to 6.1M in EXPERT, and 100M to 500M in SimPoint VLI). With 100M samples, warm-up is not an issue, at least with current cache sizes. However, with significantly smaller samples, warm-up becomes critical, especially when the sampling target accuracy is of the order of a few percent. Figure 2 illustrates this trend using sampling with 1M and 10M fixed-size intervals. Consider the 10M warm-up and 10M no warm-up bars: there is barely any accuracy difference between the two. Consider now the 1M warm-up and 1M no warm-up bars: the difference jumps to 1.7%. So for 1M samples, ignoring warm-up can wipe out the accuracy or sampling size gains.

The reason for such strict accuracy requirements is that architecture design is a trial-and-error process composed of many "micro decisions" (parameter values selection, choosing against two architecture options, etc...) based on simulation results which often correspond to small performance differences. Recent research works on sampling have similar accuracy targets [6, 7, 13].

Several recent research works [3, 4, 8] propose various techniques for reducing the warm-up size before samples while maintaining a high accuracy, but they are all separate from the sampling techniques themselves. In sampling research works, warm-up has been treated in two different ways: either using functional [14, 7] or checkpoint-based warm-up [13], or assuming warm-up is perfect based on the principle of separating warm-up and sampling issues [12, 10, 6].

Functional warm-up means the emulator is in charge of warming-up the main SRAM structures of the simulator (caches, tables), and checkpoint-based warm-up consists in storing the SRAMs state before the sample. In both cases, warm-up is performed for a given set of SRAM structures with a given set of characteristics. So, while functional and checkpointing warm-up can be very accurate when the emulated/checkpointed architecture is exactly the same as the simulated one, it is very difficult to anticipate the accuracy loss when they differ. And again, in *practice*, when a researcher is investigating many different architecture variations, that is going to happen often. For functional warm-up, there are two reasons for that to happen. First, it is sometimes difficult or just impossible to embed time-sensitive mechanisms (like many prefetching schemes) within the emulator, because it has no timing information. Second, it is also time-consuming

and impractical to adjust the emulator to every possible modification in the simulator. For checkpointing warm-up, the reasons are similar. The checkpoints correspond to a given architecture. In TurboSMARTS [13], a recent checkpointing warm-up method, the authors show how to partially relax these constraints so that the checkpoints can be reused when some architecture parameters vary, but they also acknowledge the method is difficult to adapt to some structures, such as modern branch predictors. In this article, we rely upon and advocate for traditional simulator-based warm-up, because we consider functional and checkpointing warm-ups are inappropriate for architecture researchers, who often want to explore a range of architectures and parameters without worrying about adjusting their emulator or checkpoints to each architecture, or the accuracy consequences of not doing so.

Assuming perfect warm-up for evaluating a sampling technique is also inadequate when the samples sizes are small (e.g. 1 million instructions [6]). In practice, if functional/checkpointing warm-up is not used, then the warm-up will be performed by running the simulator a few thousands to a few millions instructions before the target samples. This warm-up simulation adds up to the sample sizes and increases the overall number of simulated instructions. The warm-up size will also affect, potentially strongly, the error. In short, if warm-up is performed in the simulator, it is part of the accuracy/size tradeoff which is at the core of sampling techniques. Therefore, in order to reach the best possible tradeoff, it is unwise and potentially inefficient to treat warm-up separately from the sampling issue. BLRL [3] shows that the average warm-up size needed to decrease the error induced by warm-up to 0.43% (for 1-million instruction samples) is 453 million simulated instructions per sample; without warm-up, the error is 5.6% for the same samples.

In this article, we advocate for an integrated approach at (simulator-based) warm-up and sampling. Instead of separating warm-up and sampling, we take exactly the opposite approach, provide a common instruction *budget* for warm-up and sampling, and we attempt to spend it as wisely as possible on either one. For sampling, we try to select samples so as to minimize redundant information by relying on program-aware variable-size samples. And even within each sample, we avoid simulating redundant information. Finally we allocate simulation budget preferentially to samples that represent greater shares of the execution. For warm-up, we similarly allocate it preferentially to the most representative samples, where achieving high accuracy is key. Also, we factor in the sample size, so as not to waste warm-up budget on large samples, which need it least.

This budget and integrated approach at warm-up and sampling achieves an average CPI error of 1.68% on the 26 Spec benchmarks with an average sampling size of 288 million instructions, and at the same time, it relieves the user from any delicate decision such as setting the sampling or warm-up sizes, thanks to the integrated warm-up+sampling and the region partitioning approaches.

Section 2 presents our method for partitioning the program trace into program-aware regions. Section 3 combines this region partitioning method with a budget-based approach at distributing simulated instructions among sampling and warm-up intervals. An evaluation of the budget-based approach is presented in Section 4.

2 Program Partitioning Into Regions

In order to wisely spend the sampling/simulation budget over the whole program trace, we first decided to identify frequently repeating program regions rather than relying on fixed-size intervals. For that purpose we propose a new partitioning algorithm which is easier to deploy than the algorithms proposed by EXPERT and SimPoint VLI.

Region-Based partitioning. Our program partitioning approach is based on the principle that programs can exhibit complex control flow behavior, even within phases. More precisely, the very principle of phases means that programs usually "stay" within a set of static basic blocks for a certain time, then move to another (possibly overlapping) set of basic blocks, and so on. This set of basic blocks can span overall several parts of multiple subroutines and loops. Moreover, the order and frequency with which these basic blocks are traversed may be very irregular (think of `if` statements with very irregular behavior, think of subroutines which are called infrequently within looping statements, etc...). We call such sets of basic blocks where the program "stays" for a while **regions**. These regions capture the program *stability* while accommodating its *irregular* behavior. We propose a simple method, composed of two rules, for characterizing these basic block regions:

1. Whenever the reuse distance between two occurrences of the same basic block (expressed in number of basic blocks) is greater than a certain time T, the program is said to leave a region.

2. After the program has left a region, application of rule 1 is suspended during T basic blocks, in order to "learn" the new region.

Implicitly, we progressively build a pool of basic blocks: whenever a new basic block is accessed, we examine whether this basic block has been recently referenced

(less than T ago); if so, we assume the program is still traversing the same region of basic blocks; if not, we assume the program is leaving this region; then, the second rule gives time for the program to build the new pool of basic blocks.

SPEC	Number of Instructions	T	Num. Regions	Insn. per Region	Number of Clusters
ammp	326,548,908,728	45,000	183,558	1,778,996	49
applu	223,883,652,707	1,500	187,278	1,195,462	37
apsi	347,924,060,406	3,000	187,311	1,857,450	44
art	41,798,846,919	1,500	112,350	372,041	42
bzip2	108,878,091,744	25,000	170,903	637,075	318
crafty	191,882,991,994	100,000	199,499	961,824	527
eon	80,614,082,807	20,000	194,912	413,592	92
equake	131,518,587,184	2,000	196,991	667,637	17
facerec	211,026,682,877	35,000	196,206	1,075,536	22
fma3d	268,369,311,687	15,000	184,667	1,453,260	73
galgel	409,366,708,209	70,000	111,399	3,674,779	140
gap	269,035,811,516	90,000	192,658	1,396,442	92
gcc	46,917,702,075	20,000	95,529	4,911,357	323
gzip	84,367,396,275	30,000	170,966	493,475	167
lucas	142,398,812,356	100	187,849	758,049	56
mcf	61,867,398,195	25,000	178,469	346,653	54
mesa	281,694,701,214	80,000	187,916	1,499,046	16
mgrid	419,156,005,842	2,500	54,440	7,699,412	32
parser	546,749,947,007	300,000	177,738	3,076,157	507
perlbmk	39,933,232,781	100,000	41,866	953,834	129
sixtrack	470,948,977,898	9,500	183,823	2,561,970	46
swim	225,830,956,489	400	75,740	2,981,660	54
twolf	346,485,090,250	200,000	161,142	2,150,184	28
vortex	118,972,497,867	80,000	190,722	623,806	31
vpr	84,068,782,425	8,500	193,173	435,199	155
wupwise	349,623,848,084	200,000	13,696	25,527,442	16
Average	231,987,140,463	61,130	151,915	2,712,371	118

Table 1: *Region statistics and T.*

Since T determines which reuse distances are captured by regions, a fixed value of T can potentially miss key reuses in certain programs or conversely insufficiently discriminate regions in other programs.[1] We use a benchmark-tolerant way to capture "enough but not too many" reuses; we set T for each benchmark such that a fixed percentage P of reuse distances are captured in regions, and we experimentally found $P = 99.6\%$ would capture the appropriate amount of reuse, and thus would result in appropriate values of T for all benchmarks. Table 1 shows T and the regions statistics obtained with $P = 99.6\%$.

Sampling regions. The regions form a partition of the program trace, i.e., they define trace intervals of variable sizes. Then we can group regions based on their similarities using a clustering method. Clustering methods can group regions into clusters based on their basic block frequency characteristics, and then pick, for each cluster, one region that best represents the cluster. Unlike SimPoint which uses the *k-means* [9] clustering algorithm, we have developed *IDDCA* [11], a sampling-oriented derivative of the DCA clustering technique [1]. *IDDCA* is a dynamically adjusted clustering algorithm, which automatically decides the correct number of clusters for a given region trace. From each cluster we select the closest sample to the center of mass of the cluster as representative interval of the cluster, i.e., the interval that will be simulated to estimate the performance of the cluster samples. Table 1 shows the number of clusters identified by IDDCA from the interval traces previously generated.

[1]Note however that we did observe very good average accuracy/time trade-offs for the same T value applied across all benchmarks.

3 Integrated Sampling and Warm-Up

The general problem is to select the sampled instructions (location and number) as wisely as possible. Partitioning the program trace into regions that capture similar and recurring local behavior addresses the *location* issue. With respect to the number issue, the general philosophy should be: spend your instruction (sampling or warm-up) budget where it's most needed (and in the process, try to minimize the total needed budget).

In that spirit, we make two simple observations: (1) the weight of each cluster should be factored in when allocating its (sampling and warm-up) instruction budget, and (2) the length of each cluster representative interval should be factored in when determining the warm-up size for this interval.

Let us go back to observation (1). The goal of clustering methods, as used in BeeRS or in SimPoint, is to find a representative for each cluster of regions. Not all clusters contain the same total number of instructions; for instance, they range from 57,193 instructions to 430 billion instructions in `sixtrack` for $T = 9500$. Naturally, when extrapolating performance statistics collected for each cluster representative to the whole program trace, the relative weight of each cluster is factored in. But it also means that the performance measurement of some of the representatives will have a greater impact on the total estimated performance than others. Or, in other terms, that the performance measurement for an important representative should match as accurately as possible the average performance of the corresponding clusters. So, we should allocate a greater share of the simulated instruction budget to representatives of large clusters in order to more accurately estimate their performance. The number of simulated instructions allocated per region consists of the region size plus the additional instructions simulated

for warm-up purposes. Which brings observation (2). If a cluster representative interval is large (the representative itself, not necessarily the cluster), then it will need less warm-up instructions as the start-up effect will be diluted in the simulation of the cluster representative interval. Conversely, small representative intervals need significant warm-up, which is a key reason why SMARTS and EXPERT use continuous emulator/checkpointing-based warm-up.

Determining sampling and warm-up size. Let us call B the total instruction budget, i.e., the maximum number of simulated instructions (including warm-up). Let us number clusters i, with $1 \leq i \leq k$, where k is the total number of clusters, and let us call S_i the total size (in number of instructions) of cluster i; the clusters are ordered by decreasing size, i.e., $S_i > S_j$, if $i < j$. f_i is the weight factor of cluster i over the whole program trace size ($f_i = \frac{S_i}{\sum_{r=1..k} S_r}$), and s_i is the size of the representative interval of cluster i.

Based on observation (1), we distribute the budget for each cluster based on the global weight f_i of the cluster. For that purpose, we define B_i as the maximum simulation budget for cluster i (sampling and warm-up); $B_1 = B \times f_1$ and $B_i = (B - \sum_{j=1..i-1} B_j) \times \frac{f_i}{\sum_{l=i..k} f_l}, \forall\, i > 1$, which can be simplified to $B_i = B \times f_i$ if all the clusters are considered, i.e. $\sum_{i=1..k} f_i = 1$. The actual number of simulated instructions for cluster i is $r_i + w_i$ where r_i is the sampling size (it is a subset of the representative of cluster i), and w_i is the warm-up size.

Since the sampling size r_i must be smaller than the budget B_i, i.e., $r_i = \min(s_i,\ B_i)$, we sometimes need to truncate the simulation of the cluster representative. It rarely degrades accuracy, thanks to the looping behavior which is at the core of our region-partitioning scheme. In fact, we may often truncate more this simulation, providing opportunities for further simulation time reductions.

Based on observation (2), we preferably allocate warm-up instructions to small samples, within the constraint of budget B_i, i.e., $w_i = B_i - r_i$. The warm-up instructions w_i are instructions preceding the representative of cluster i. Now, due to our region-based partitioning approach, these instructions may reference code sections and data structures which are distinct from the ones referenced in the representative. To avoid simulating useless warm-up instructions, we use the BLRL [3] (*Boundary Line Reuse Latency*) technique for determining the size of the useful warm-up interval. BLRL consists in collecting the memory addresses and branch instruction addresses used in the sampled interval, and to identify the earliest point in the trace before the interval where they will be all accessed.

Instruction Cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Predictors	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 8 operations per cycle, 64 entry re-order buffer
Memory Disambiguation	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Functional Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 2: *Baseline simulation model.*

By starting the warm-up at that point most SRAM structures are likely to be adequately warmed-up (e.g., the first access to an address will be correctly identified as a hit or a miss) independently of the SRAM structures sizes. However, under that constraint, the actual warm-up interval per sampled interval can be very large (e.g., `parser` requires more than 2 billion warm-up instructions for a region of only 1.8 million instructions), so the authors propose to set a percentage threshold of the sampled interval addresses covered in the warm-up interval, thereby relaxing the constraint and significantly reducing the warm-up interval size (we use a threshold of 95%). Still, because the BeeRS budget approach introduces a size constraint on the warm-up interval, we slightly modify BLRL by adding an instruction threshold, i.e., the w_i previously computed, to the 95% percentage threshold (we take the smallest of the two warm-up intervals).

4 Evaluation

For evaluation purposes, we used the SimpleScalar [2] 3.0b toolset for the Alpha ISA and experimented with all 26 SPEC CPU2000 benchmarks. To create the regions we used the *sim-fast* emulator. Table 2 shows the microarchitecture configuration used for our experiments.

Figures 1 and 2 respectively show the number of instructions and accuracy of different BeeRS and SimPoint configurations (the maximum number of samples is set to 50 for 10M intervals, and to 100 for 1M intervals, so as to provide a fair accuracy/size comparison with BeeRS). We use perfect warm-up for SimPoint as in most of the

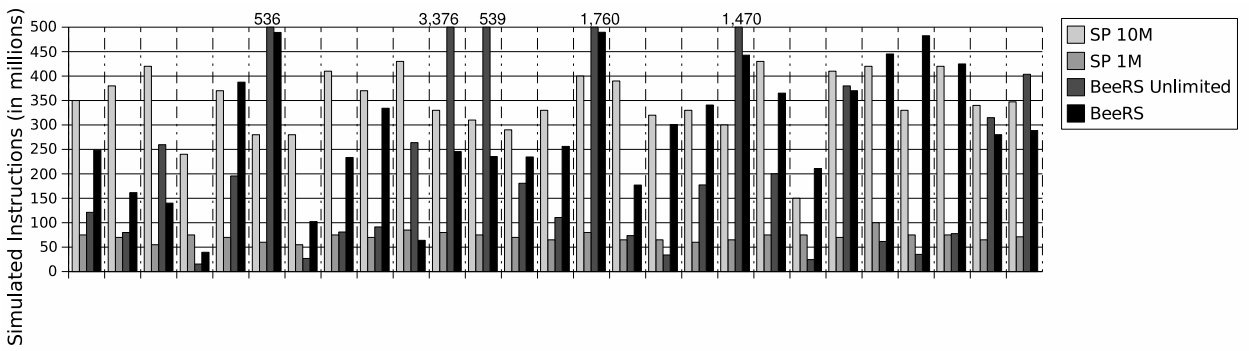


Figure 1: *Number of simulated instructions with different sampling techniques.*

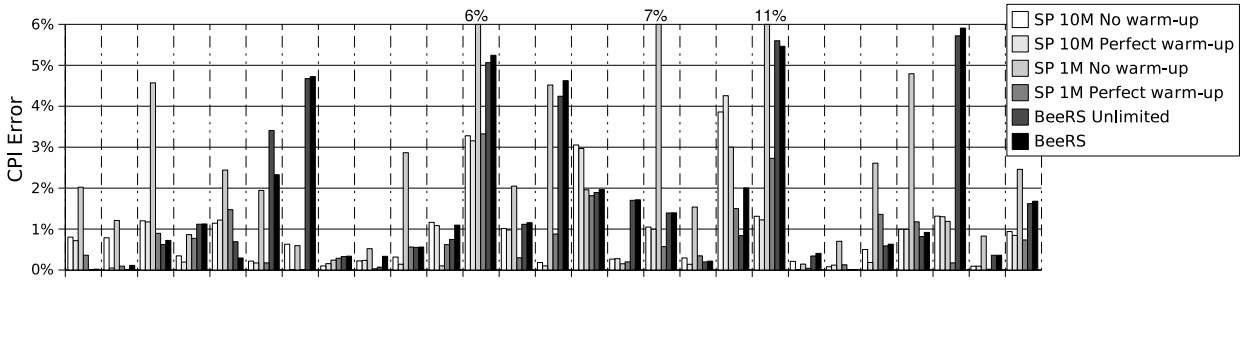


Figure 2: *CPI error.*

articles [12, 10, 6, 5] (recall SimPoint treats sampling as an issue independent from warm-up). As mentioned in the introduction, while the accuracy of SimPoint 10M is barely sensitive to warm-up, SimPoint 1M becomes fairly sensitive (from 0.7% down to 2.4%), and the trend can only worsen as the sample size decreases. Therefore, while SimPoint 1M requires little instructions compared to SimPoint 10M or BeeRS with warm-up, it would actually need to spend additional budget on warm-up in order to preserve its accuracy. BeeRS has lower accuracy but requires fewer instructions than SimPoint 10M. More importantly, the user never needs worry about setting the appropriate sample and warm-up sizes for a new given program, it is all integrated in the partitioning and budgeting approach. All the user needs to set/decide is the maximum simulation budget (i.e., time).

We also evaluated BeeRS without any budget limitation, see `BeeRS Unlimited`. We can see that wisely allocating the budget allows drastic reductions of the number of simulated instructions with limited impact on accuracy (from 1.62% to 1.68%). Note that the same allocation strategies also enable to use significantly less than the maximum budget, i.e., 288 million instead of 500 million instructions. Figure 3 displays for each benchmark, how BeeRS actually distributes its instruction budget between sample and warm-up. Obviously, the number of simulated instructions devoted to sampling is rather low (only 84 million instructions on average). This value is close to the number of instructions simulated by SimPoint with 1 million instructions samples (71 million instructions). As a correlate, it is also clear from this figure that warm-up is highly instruction-consuming, as it accounts on average for roughly 70% of the total simulated instuctions.

5 Conclusions

The rationale for BeeRS is that some of the most recent and efficient sampling techniques have implicit applicability restrictions due to their warm-up approach (in the emulator or by checkpointing architectural states, or simply using perfect warm-up on the principle of separating sampling and warm-up issues), which can make it difficult to explore specific and/or a large range of architectural optimizations, specially when the simulated intervals are small. BeeRS makes no compromise on applicability, and achieves an accuracy/time tradeoff that is of the same

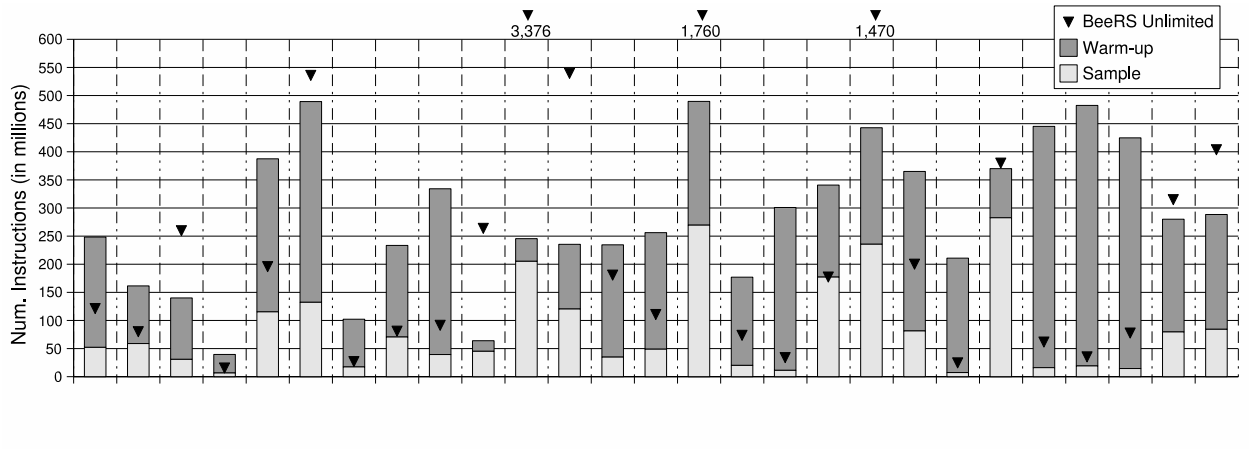


Figure 3: *Distribution of the number of sampled and warmed-up instructions with BeeRS.*

order of the best sampling techniques. The key features of BeeRS is a novel definition of sampling intervals, and an integrated budgeted approach at distributing simulation instructions among sampling and warm-up intervals, depending on how they can best benefit to accuracy.

References

[1] A. Baune, F. T. Sommer, M. Erb, D. Wildgruber, B. Kardatzki, G. Palm, and W. Grodd. Dynamical Cluster Analysis of Cortical fMRI Activation. In *NeuroImage 6(5)*, pages 477 – 489, May 1999.

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

[3] L. Eeckhout, S. Eyerman, B. Callens, and K. De Bosschere. Accurately Warmed-up Trace Samples for the Evaluation of Cache Memories. In I. Banicescu, editor, *Proceedings of the High Performance Computing Symposium - HPC2003*, pages 267–274, Orlando, FL, USA, 4 2003. SCS.

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

[5] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. *ISPASS '05: IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.

[6] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for Phase Classification. *ISPASS '04: IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[7] Wei Liu and Michael C. Huang. EXPERT: expedited simulation exploiting program behavior repetition. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 126–135. ACM Press, 2004.

[8] Yue Luo, Lizy K. John, and Lieven 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.

[9] Dan Pelleg and Andrew W. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734. Morgan Kaufmann Publishers Inc., 2000.

[10] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 244. IEEE Computer Society, 2003.

[11] Daniel Gracia Pérez, Hugues Berry, and Olivier Temam. IDDCA: A New Clustering Approach For Sampling. In *MoBS '05: Workshop on Modeling, Benchmarking and Simulation*, 2005.

[12] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, 2002.

[13] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. *SIGMETRICS '05*, June 2005.

[14] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James 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.