
A SAMPLING METHOD FOCUSING ON PRACTICALITY

THIS SAMPLING TECHNIQUE, WHICH IS HARDWARE-INDEPENDENT AND ALMOST ENTIRELY TRANSPARENT TO THE USER, EMPLOYS A BUDGET-BASED APPROACH THAT JOINTLY CONSIDERS WARM-UP AND SAMPLING COSTS, PRESENTS THEM AS A SINGLE PARAMETER TO THE USER, AND DISTRIBUTES SIMULATED INSTRUCTIONS BETWEEN WARM-UP AND SAMPLING BASED ON REGION PARTITIONING AND CLUSTERING INFORMATION. ALTHOUGH IT FOCUSES ON PRACTICALITY, THE TECHNIQUE DELIVERS NEARLY STATE-OF-THE-ART SPEED AND ACCURACY.

Daniel Gracia Pérez
CEA LIST, France

Hugues Berry
Olivier Temam
INRIA Futurs, France

..... In the past few years, research has demonstrated that sampling can dramatically speed up architecture simulation, and several sampling techniques have already come into wide use. However, for a sampling technique to be both easily and properly used—plugged in and used reliably with many simulators and with little or no user effort or knowledge—it must fulfill several conditions: It should require no hardware-dependent modification of the functional or timing simulator; it should simultaneously handle warm-up and sampling; and it should deliver high speed and accuracy. The advent of generic and modular simulation frameworks such as Asim, SystemC, LSE, MicroLib, and UniSim brings additional requirements: A sampling technique should also be simulator independent and almost entirely transparent to the user. (The sidebar, “Modular simulation and sampling,” explains these frameworks in more detail.)

In this article, we explain how even the

most successful of the existing sampling techniques fall short on practicality, especially when it comes to warm-up. We propose a more practical alternative: a sampling technique that focuses specifically on transparency, while also delivering nearly state-of-the-art speed and accuracy. The technique offers a hardware-independent, integrated approach to warm-up and sampling that requires no modification of the functional simulator and relies solely on the performance simulator for warm-up. Our technique makes three major contributions to the sampling field:

- a technique for splitting an execution trace into a potentially very large number of variable-size regions to capture the program’s dynamic control flow;
- a new clustering method capable of efficiently coping with so many regions, and
- a budget-based method for jointly considering warm-up and sampling costs,

presenting them as a single parameter to the user, and distributing the number of simulated instructions between warm-up and sampling based on the region partitioning and clustering information.

Overall, our sampling method achieves a trade-off of accuracy and time close to the best reported results using clustering-based sampling—which usually assume perfect warm-up or require hardware-dependent warm-up. For the SPEC benchmark suite, our technique's average cycles per instruction (CPI) error is 1.68 percent, and it simulates an average of 288 million instructions per benchmark. The technique and tool can be readily applied to a wide range of benchmarks, architectures, and simulators, and it will be a sampling option within the UniSim modular simulation framework.

The sampling trade-off

To be efficient and useful for architecture researchers, a sampling technique must balance simulation accuracy, overall simulation time, and practicality, a characteristic that includes the range of architectures a technique can target, user effort, and transparency. SimPoint sparked a surge of interest in sampling because the technique is both efficient and easy to use.¹ Considering the achievements of SimPoint and the techniques and improvements that followed,²⁻⁵ do we need to push sampling research any further?

After surveying existing sampling techniques, we've concluded that, despite significant recent progress, they do not achieve a satisfactory trade-off of accuracy, speed, and practicality. Before we explain why in detail, let's define satisfactory goals for these three characteristics.

Accuracy

Designing an architecture mechanism is a trial-and-error process composed of many microdecisions—selecting parameter values, choosing between two architecture options, and so on. Designers base these decisions on simulation results that often correspond to small performance differences. Thus, a good target for sampling accuracy is a *CPI error rate* (the percent difference between the cycles per instruction obtained over the whole program

Modular simulation and sampling

Most sampling techniques have been applied only to a specific simulator. However, modular simulation frameworks—such as Asim,¹ SystemC, the Liberty Simulation Environment,² MicroLib,³ and UniSim—use a generic engine that calls different simulator modules from a library. These frameworks let designers rapidly change architecture components by plugging in alternative modules. Achieving this flexibility, however, means sacrificing speed: Modular simulators are typically 10 to 20 times slower than monolithic simulators such as SimpleScalar, so sampling is critical for them. And, because any sampling technique would have to plug directly into the generic engine, it would have to be simulator independent.

Plugging a sampling technique into the engine would bring the benefit of automatically providing the sampling capability to almost any simulator written for that framework. However, because the engine would be used to simulate many different architectures, for transparent use the sampling technique would have to be architecture independent as well.

Some modulator simulation framework Web sites include

- Liberty Simulation Environment, <http://liberty.princeton.edu/Software/LSE/>;
- MicroLib, <http://microlib.org/>;
- SystemC, <http://www.systemc.org/>; and
- UniSim, <http://unisim.org/>.

References

1. J.S. Emer et al., "Asim: A Performance Model Framework," *Computer*, vol. 35, no. 2, Feb. 2002, pp. 68-76.
2. M. Vachharajani et al., "Microarchitectural Exploration with Liberty," *Proc. Int'l Symp. Microarchitecture (Micro 35)*, IEEE CS Press, 2002, pp. 271-282.
3. D. Gracia Pérez, G. Mouchard, and O. Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms," *Proc. Int'l Symp. Microarchitecture (Micro 37-04)*, IEEE CS Press, 2004, pp. 43-54.

execution and over the sampled intervals) on the order of one or a few percent.

Speed

For many sampling techniques, overall simulation time—that is, the sum of the simulation time of the sampling intervals plus functional simulation between the intervals—is dominated by the functional simulation time. This is because the sampling intervals correspond to only a few percent of the total number of instructions in the program trace. In this context, improving sampling efficiency by reducing the total sample size would bring little improvement.

However, recent studies show that techniques using checkpointing can drastically reduce overall simulation time—from a few hours to a few minutes per benchmark—by getting rid of functional simulation. (TurboSmarts and SimPoint with load value

sequence and memory hierarchy state use this approach.^{5,6}) Considering the converging factors of slower modular simulation and increasingly complex processor architectures (more complex superscalar processors and multi-cores), checkpointing will probably soon be a necessary complement to sampling techniques.

Assuming checkpointing does become a common feature, overall simulation time will rest entirely on the number of simulated instructions. In other words, reducing the sampling size by x percent will reduce the overall simulation time by x percent as well. Because this article is solely concerned with sampling technique (not functional simulation versus checkpointing), and because the sampling technique we propose is perfectly compatible with checkpointing, throughout this article we use *time* to refer to the number of simulated instructions (rather than overall simulation time). Thus, we focus on reducing the total number of simulated instructions.

Practicality

Practicality could be the least measurable characteristic of the sampling trade-off, and researchers have paid it the least attention. However, in the end, scope and ease of use are just as important as efficiency in enticing users to adopt a new methodology. Given that architecture researchers have long been using the crude approach of randomly picking traces of arbitrary sizes, it's safe to assume that they will discard any technique that is not simple to use or that imposes too many restrictions. The rationale of this article is to achieve a good trade-off of accuracy and time without degrading practicality.

How current techniques lack practicality

Arguably, the current best sampling techniques are SimPoint,⁷⁻⁹ Smarts and TurboSmarts,^{2,5} and Expert.³ These techniques exhibit the two possible approaches for selecting sampling intervals: using a large number of uniformly (or randomly) selected small intervals, and using a few large and carefully selected intervals.

Smarts, which adopts the first approach and uses uniform sampling with a large number of tiny intervals, achieves one of the best trade-offs of accuracy and time: 0.64 percent CPI error and 50 million instructions per bench-

mark on the SPEC suite using SimpleScalar.¹⁰ However, because of the small size of the intervals (around 1,000 instructions), it must continually warm the main SRAM structures in the functional simulator (especially the caches, and possibly the branch predictors), a process called functional warm-up. The main limitation of this approach is not efficiency but practicality: A range of cache mechanisms—for instance, prefetching mechanisms—do not lend themselves to this warm-up approach. Prefetching naturally affects cache behavior, so the functional simulator should warm the prefetching mechanism as well, but this requires timing information, which the functional simulator does not have. Moreover, with this approach to warm-up, whenever an architecture optimization affects a large processor SRAM structure—a cache, a branch predictor, a translation look-aside buffer (TLB), and so on—the structure should ideally be implemented in both the functional and timing simulators. For some mechanisms, such as prefetching, this is fairly impractical, if not impossible. Even when it is possible, this warm-up requirement places a significant software-engineering burden on the user.

However, the Smarts researchers recently proposed embedding the warm-up information in the simulator modules of their Flexus infrastructure to reduce that burden (<http://www.ece.cmu.edu/~simflex/flexus.html>). The same authors also proposed TurboSmarts, which obviates functional warm-up by checkpointing microarchitectural state, such as the content of SRAM, DRAM, and register structures. Although this method drastically improves overall simulation time by eliminating full-program functional simulation, it has practicality restrictions related to architecture dependence. The authors show how to partially relax these constraints so that the checkpoints can be reused when some architecture parameters vary, but they acknowledge that the method is difficult to adapt to some structures, such as modern branch predictors.

Still more recently, Barr et al. have proposed compressing all the trace information required for warming branch predictors;¹¹ this approach makes it possible to tackle a wider range of branch predictors, but it is still specific to that type of architecture component.

Expert adopts the second sampling approach—judiciously picking a few intervals of different sizes.³ In Expert, the selection is based on characteristic program constructs such as loops or subroutines. This program-aware approach to interval selection brings excellent accuracy: for SPEC benchmarks and SimpleScalar, averages of 0.89 percent CPI error and 1 billion simulated instructions. (This second figure is an approximation derived from the article's figures.³) However, because of the small interval granularity, or size, this technique, like Smarts, must resort to continual warm-up in the functional simulator—with the same practicality limitations. Expert achieves further gains in accuracy and time by precharacterizing sampling intervals using simulations; but, again, this approach raises significant practicality and stability issues when the architecture varies. Like TurboSmarts, Expert uses checkpointing to drastically reduce simulation time, but this is also based on microarchitectural state (for caches and branch predictors). In other words, this technique requires architecture-dependent warm-up.

The original version of SimPoint was the first step toward the practice of wisely picking sampling intervals according to basic block frequency characteristics.¹ However, having few but large intervals achieved a poorer trade-off of accuracy and time than Smarts and Expert did later: a CPI error of 3.7 percent and 500 million instructions per program, again with the same benchmarks and simulator. Although most articles about SimPoint assume perfect warm-up (implemented using functional warm-up),^{1,8,9} the original sampling interval sizes, 100 million instructions, were large enough that the technique could obtain good accuracy without warm-up. Our own SimPoint experiments confirm that warm-up has little impact with such large traces.

In the past few years, the SimPoint group has experimented with more but smaller intervals (10- and 1-million instruction intervals and a maximum of 300 simulation points^{8,9}) to achieve a better trade-off of accuracy and time. The group recently proposed a variable-length interval technique similar to that in Expert, but with a different interval selection approach.⁴ Although this technique exhibits very good accuracy (about a 2.5 percent CPI error over nine SPEC benchmarks), the simu-

lation time is still long, about 1 billion instructions. The study assumes perfect warm-up before the sampled intervals, which makes it difficult to evaluate the technique's actual trade-off of accuracy, time, and practicality.

Budget-based sampling

The sampling technique we've devised focuses mainly on practicality. It takes a holistic approach, considering sampling (trace partitioning), clustering (sample selection), and warm-up together. We set a goal of achieving accuracy similar to that of Smarts and TurboSmarts, Expert, and SimPoint VLI—on the order of one or a few percent CPI error—and then sought to minimize the total number of simulated instructions (warm-up plus measurement). Although our number of simulated instructions is higher than that for Smarts and TurboSmarts, our technique is more practical: Because it relies strictly on the timing simulator for warm-up—instead of the functional simulator or checkpointing—there is no need to implement warm-up for caches, TLBs, or branch-prediction tables in the functional simulator, or to include microarchitectural state in the checkpoints.

Our variable-size partitioning technique, new clustering technique, and budget-based method for distributing the simulation time between warm-up and performance measurement all contribute to our technique's efficiency and practicality.

Partitioning the program into regions

The main benefit of using variable-size regions is that it allows decomposition of the program trace into a very large number of regions. Smaller regions are better for sampling accuracy, but they have two drawbacks: they are more sensitive to warm-up, and they increase the number of regions, straining clustering techniques. Our method avoids both drawbacks: Some of the regions are large and thus less sensitive to warm-up, and the presence of these large regions keeps the total number of regions reasonable.¹² In some sense, our method performs a kind of program-aware preclustering—as if it had split the trace into very small regions and then clustered them into larger ones, but without incurring the high cost of clustering a huge number of very small regions. Recently, Expert and

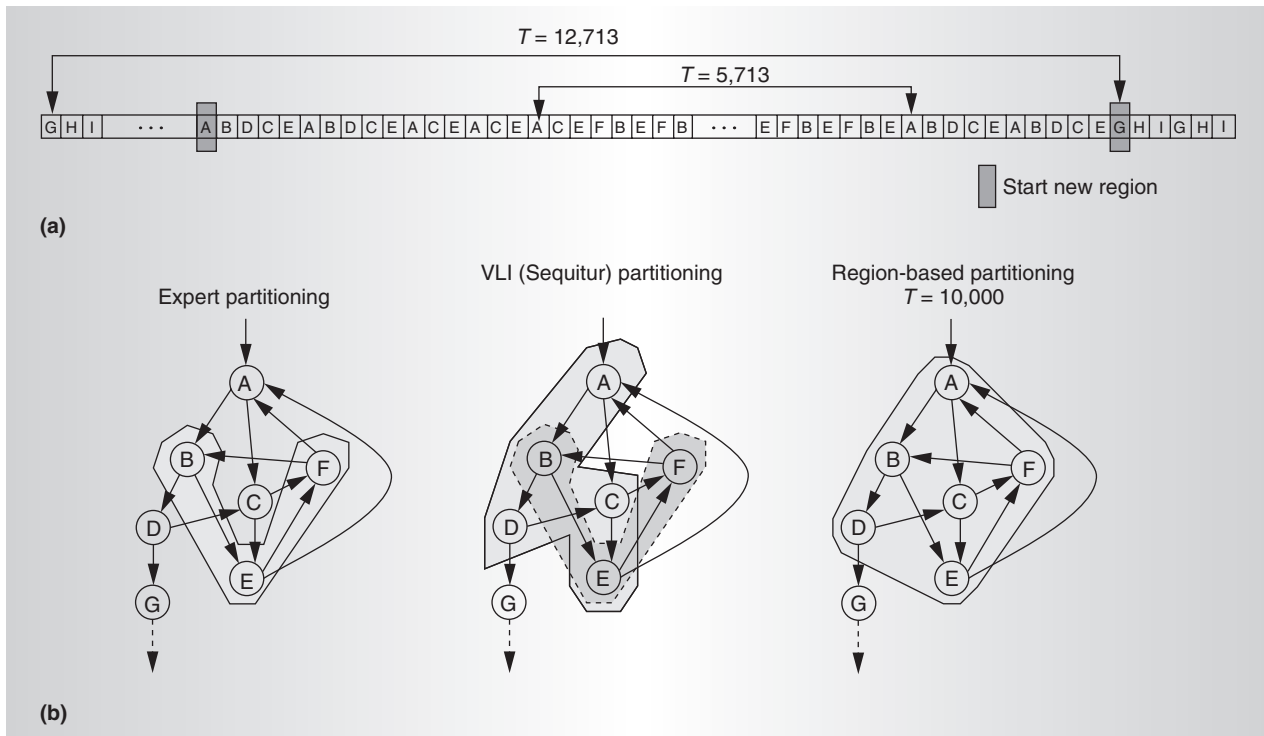


Figure 1. Program trace partitioning algorithms.

SimPoint VLI explored variable-size regions with positive accuracy results. However, for several reasons, we decided to retain neither the Expert nor the SimPoint VLI program partitioning methods.

Expert partitioning

The principle of Expert is to partition the program based on subroutines, distinguishing among long, short, and infrequently executed subroutines; characterize the performance variability of these subroutines using simulation; and then select the number and location of subroutine representatives. Beyond the practicality problems related to continuous warm-up, this partitioning and characterization method has two flaws: The characterization is hardware dependent, and it heavily relies on loops. Hardware-dependent characterization means a code must be simulated on an architecture before it can be sampled on that architecture. If the target architecture changes, it is hard to anticipate the consequences for the characterization. Loop-based subroutine characterization can also be problematic for code with complex control flow behavior—multiple *if* statements within a procedure, recursion, and so on. Not

surprisingly, Expert demonstrates better results with the SPEC floating-point benchmarks than with the SPEC integer benchmarks.

Consider the example in Figure 1, which depicts a sequence of basic blocks within the program static control flow graph: Each lettered node denotes a basic block. Assuming BEF is a large loop called within an even larger loop ABDCE, Expert would most likely define an interval that encapsulates only BEF; multiple invocations of the BEF loop would breed multiple intervals.

SimPoint VLI partitioning

SimPoint VLI adopts a different approach, although it is also based on loops and procedures. SimPoint VLI numbers each code structure and then views the trace as a sequence of identifiers. Then, the Sequitur hierarchical pattern-matching algorithm identifies repeating sequences of variable sizes within the trace.^{13,14} The main problem with this approach is that it relies on an exact match between two sequences within the trace. Programs with complex control flow due to non-trivial *if* statement behavior can exhibit many different sequences or might only enable exact

matching for smaller sequences. In the example in Figure 1, Sequitur would partition the sequence into two main recurring parts: ABDCE and BEF. After this pattern-matching phase, the SimPoint VLI technique applies several heuristics to relieve this exact match constraint to obtain longer sequences, improving the flexibility of this type of sequence partitioning.

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 small code sections such as loops or several subroutines. Moreover, the order and frequency with which the program traverses these basic blocks can be very irregular—encountering, for example, *if* statements with very irregular behavior, subroutines that are called infrequently within looping statements, and so on. We call such sets of basic blocks where the program stays for a time *regions*. These regions capture the program’s stability while accommodating its irregular behavior. We propose a simple method, consisting of two rules, for characterizing these basic block regions:

- 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.
- After the program has left a region, the algorithm suspends application of rule 1 during T basic blocks, for the method to “learn” the new region.

Implicitly, the method progressively builds a pool of basic blocks: Whenever the program accesses a new basic block, our algorithm examines whether this basic block has been recently referenced (less than T ago). If so, it assumes the program is still traversing the same region of basic blocks; if not, it assumes the program is leaving the region. Then, the second rule gives time for the program to

build the new pool of basic blocks. In Figure 1, because the program references A, B, C, D, E, and F within a time interval smaller than T , they all belong to the same region, despite the sometimes irregular control flow behavior. G, H, and I mark the beginning of a new region because their reuse distance is greater than T .

Because T determines which reuse distances are captured by regions, a fixed value of T might miss key reuses in certain programs or, conversely, insufficiently discriminate regions in other programs. (However, we did observe very good average accuracy and time trade-offs when we applied the same T value across all benchmarks.) We use a benchmark-tolerant method to capture “enough but not too many” reuses. This method sets T for each benchmark so that the algorithm captures a fixed percentage P of reuse distances in regions. Experimentally, we found a P of 99.6 would capture the appropriate amount of reuse, thus resulting in appropriate values of T for all benchmarks. Table 1 shows T and the region statistics obtained with P set to 99.6.

For some programs, the average region size is on the order of a few hundred thousand instructions; in crafty, some regions are as small as a few thousand instructions. Thanks to a mix of large and small regions in each program, the total number of regions is not excessively high: several tens of thousands to a few hundred thousand. However, it is large enough that the k -means clustering method used in SimPoint¹⁵ (for fixed-size, 1-million-instruction intervals) would take an excessively long time, on the order of a day per benchmark. Our clustering method can reduce clustering time by more than two orders of magnitude.

Clustering many regions using IDCCA

In describing our clustering method, we use the term *region* synonymously with the more classic term *interval*. As proposed in the SimPoint literature, we define the distance between two intervals as the distance between their two basic-block vectors.¹

The popular k -means clustering technique has three main shortcomings:

- The method works by randomly selecting intervals at the start-up phase, so that several runs of the method on the same

trace can provide different sampling intervals and thus different accuracy results.

- The number of clusters (k) is a user-selectable parameter, but it is sensitive to benchmarks or traces, so that inappropriately

setting this parameter can degrade either simulation time or accuracy.

- The method requires multiple passes, which might be impractical with a large number of intervals.

Table 1. Region statistics and values of T .

SPEC program	No. of instructions	T	No. of regions	Instructions per region	No. 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

To evaluate execution time, we ran the default SimPoint clustering script (runsimpont, with default parameters except $\max k = 100$) on our region partitioning. On an Athlon XP 2800+, the k -means technique requires 21 hours per benchmark (up to two days for crafty). In comparison, our clustering technique, interleaved double dynamical clustering analysis (IDDCA),¹⁶ took 9 minutes on average and 44 minutes for crafty. Figure 2 details the comparison.

IDDCA algorithm

We derived IDDCA from the dynamical clustering analysis (DCA) method and adapted it to sampling.¹⁷ IDDCA is an online algorithm, clustering regions one at a time, constantly refining the number of clusters and their centroids. This dynamic process relies upon three different parameters: θ_{new} , θ_{merge} , and $\theta_{\text{step factor}}$.

Intuitively, θ_{new} , θ_{merge} , and $\theta_{\text{step factor}}$ control the creation

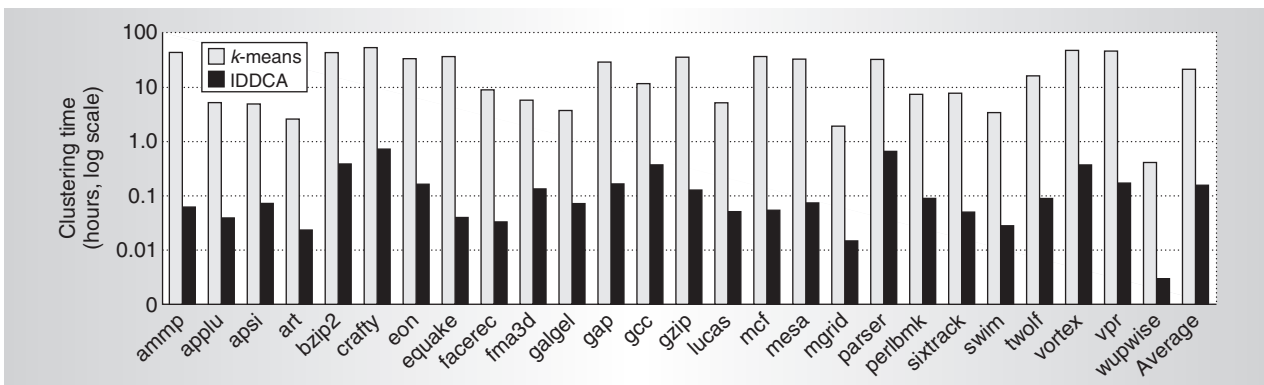


Figure 2. Clustering time of IDDCA versus k -means (logarithmic scale).

and merging of clusters. θ_{new} and θ_{merge} are threshold distance parameters for determining when a point is far enough from other clusters to induce the creation of a new cluster, or close enough to an existing cluster to be merged into it. $\theta_{\text{step factor}}$ determines the rate at which these threshold distances change. θ_{new} and θ_{merge} are initialized using a simple heuristic: 10 percent of the distance between the global centroid (the centroid of all regions) and the farthest region. $\theta_{\text{step factor}}$ is related to the number of data points, but the clustering method is robust enough to tolerate the same $\theta_{\text{step factor}}$ value across all SPEC benchmarks; we set it empirically to 10^{-5} .

IDDCA starts with two elements:

- an empty cluster list, and
- the list of regions to cluster (called R).

The regions in R are regularly interleaved to make the online clustering method less sensitive to the original program trace order. Let us assume there are N regions in the trace and the interleaving factor is I ; then the list is

1, $N/I + 1$, $2N/I + 1$, ..., $[(I-1) \times N/I] + 1$, 2, $N/I + 2$, $2N/I + 2$, ..., $[(I-1) \times N/I] + 2$, ...

The method is fairly insensitive to interleaving for $I \geq 2$, and we selected $I = 10$ for all benchmarks. Randomly picking regions would have performed similarly well or better; because of implementation constraints, we did not use this method.

Then, the IDDCA algorithm is as follows:

1. Pick a region r from the list of regions R ; if there is no cluster yet, create a first cluster containing region r and go to step 5.
2. Find the cluster c_i with the closest centroid to the current region r and compute the distance d between r and the centroid of c_i .
3. If d is greater than θ_{new} , create a new cluster containing the current region r .
4. If d is less than or equal to θ_{new} :
 - Add r to cluster c_i and update c_i centroid accordingly.
 - Find the cluster c_j with the closest centroid to that of c_i . If the distance between the centroids of c_i and c_j is less than or equal to θ_{merge} , merge the

two clusters into a unique one and compute its centroid.

- Update θ_{new} and θ_{merge} thresholds to make cluster creation and merger more difficult. For this purpose, increase θ_{new} and decrease θ_{merge} as follows: $\theta_{\text{new}} = \theta_{\text{new}} / (1 - \theta_{\text{step factor}})$ and $\theta_{\text{merge}} = \theta_{\text{merge}} (1 - \theta_{\text{step factor}})$.
5. Remove r from the list of regions R . If there are no more regions in R , then the process terminates; otherwise, go to step 1.

At the end of this process, IDDCA has created a set of clusters. If one of the clusters contains more than 90 percent of the regions, we apply IDDCA again hierarchically within this cluster; until clustering is spread out enough so that no cluster accounts for 90 percent or more of the regions. Finally, the instructions that must be simulated (sampled) are the region individuals closest to each cluster's centroid, one per cluster.

Weighted vs. unweighted IDDCA

Because large regions represent a greater share of the global execution trace than small regions, regions should normally be weighted with their size for computing the centroid. SimPoint VLI, for example, weights intervals with their size when applying k -means. However, we decided *not* to weight regions with their size, so that our method would give precedence to sampling size reduction over accuracy. We made this decision because initial sampling results suggested that we needed to focus more effort on size than on accuracy.

Nonetheless, to investigate the effect of not weighting the clusters, we ran IDDCA clustering with weighted clusters as well. As expected, this increased the total sampling size, and rather significantly—by 34 percent, as Figure 3 shows. More surprisingly, the accuracy is lower with weighted clusters than with unweighted clusters: 2 percent CPI error for weighted clusters versus 1.62 percent for unweighted clusters, as Figure 4 shows. Combined, these two observations empirically validate the unweighted approach.

The reason unweighted clustering performs so well is related both to the size distribution of intervals within a cluster and to warming. Within a single cluster, interval size can vary significantly, even between intervals approximately

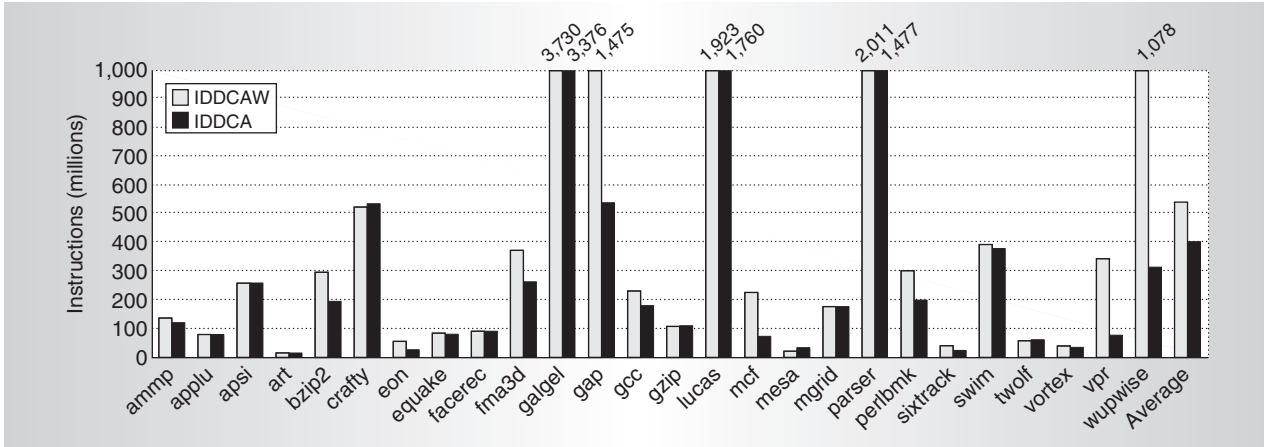


Figure 3. Number of simulated instructions with IDDCA and weighted IDDCA (IDDCAW).

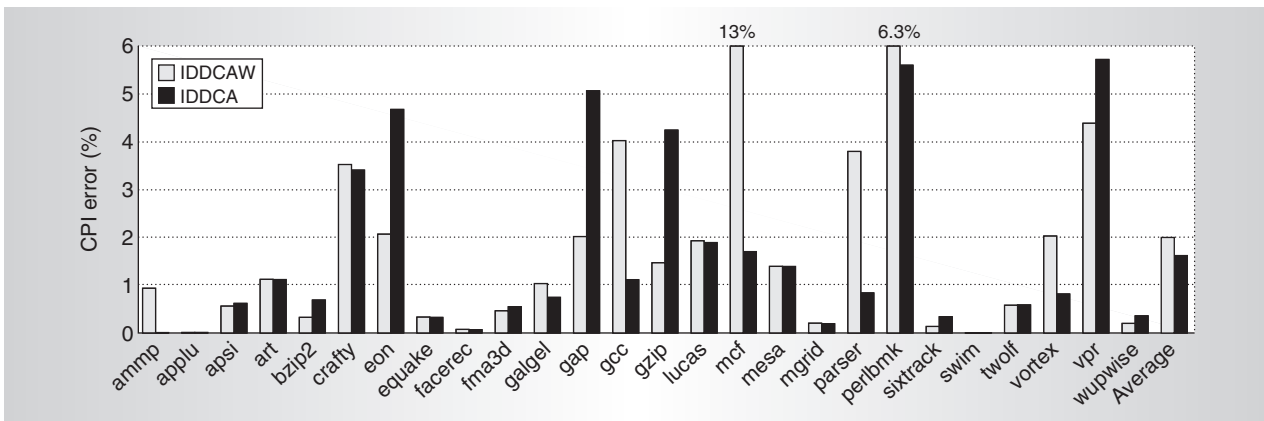


Figure 4. Simulation accuracy with IDDCA and weighted IDDCA.

the same distance from the centroid. As a result, sometimes it is possible to drastically reduce the representative interval size without significantly affecting accuracy. Moreover, the results in Figure 3 include warming, because we wanted to evaluate the best clustering strategy for our global method. Because our method uses a fixed budget distributed between sampling and warming, any simulated instruction budget not consumed by sampling can be spent on warming. Thus, the method's overall accuracy improves when it selects smaller sampling-interval representatives. This property illustrates the benefits of proper integration of the different components of a sampling strategy.

Budget-oriented integration of warm-up and sampling

Because our method implements warm-up using the main simulator (as opposed to the

functional simulator or checkpointing), warm-up and performance measurement share the same simulation budget. Because simulation is costly, we must take great care to wisely allocate the simulated instructions. And because of both variable-size regions and warm-up implemented through simulation, we must determine the budget allocated to each cluster (our method simulates one region representative per cluster). Our general philosophy is to spend the budget where it's most needed—and simultaneously minimize the total needed budget. In that spirit, we make two simple observations:

1. Each cluster's weight should be a factor in the allocation of its warm-up and measurement instruction budget; we define the cluster weight as the total number of trace (dynamic) instructions in the clus-

ter, which is itself the sum of the lengths of all intervals in the cluster.

2. The length of each cluster representative region should factor into the determination of the warm-up size for this region.

Regarding the first observation, the goal of clustering methods such as IDDCA and k -means is to find a representative for each cluster of regions. Not all clusters contain the same total number of instructions; for instance, in sixtrack, the cluster sizes range from 57,193 instructions to 430 billion instructions. Naturally, when extrapolating performance statistics collected for each cluster representative to the entire program trace, sampling methods factor in each cluster's relative weight. Unlike the weighting for the clustering method, this weighting affects only accuracy, not size. Weighting means that the total estimated performance is more influenced by the performance measurements of some representatives than others. So, we should allocate a greater share of the simulated instruction budget to representatives of large clusters, to more accurately estimate the performance of these strong influences.

The number of simulated instructions allocated per region consists of the region size plus the additional instructions simulated for warm-up, which brings us to our second observation. If a cluster representative region is large (the representative itself, not necessarily the cluster), it will need fewer warm-up instructions because the lengthy simulation will dilute the start-up effect. Conversely, small representative regions need significant warm-up, which is a key reason that Smarts and Expert use continuous functional simulator or checkpointing-based warm-up.

Now let's look at how our technique determines measurement and warm-up sizes. We call B the total instruction budget—that is, the maximum number of simulated instructions, including warm-up. We number clusters i , with $1 \leq i \leq k$, where k is the total number of clusters. S_i is the total size (in number of instructions) of cluster i ; the clusters are ordered by decreasing size—that is, $S_i > S_j$ if $i < j$. The weight factor of cluster i over the entire program trace size is

$$f_i = \frac{s_i}{\sum_{r=1..k} s_r},$$

and s_i is the size of the representative region of cluster i .

Because of observation 1, we distribute the budget for each cluster based on that cluster's global weight f_i . We define B_i as the maximum simulation budget for cluster i (measurement and warm-up). $B_1 = Bf_1$ and

$$B_i = \left[B - \left(\sum_{j=1..i-1} B_j \right) \right] \frac{f_i}{\sum_{l=i..k} f_l}, \quad \forall i > 1.$$

We can simplify this to $B_i = Bf_i$ if all the clusters are considered—that is, if

$$\sum_{i=1..k} f_i = 1.$$

The actual number of simulated instructions for cluster i is $r_i + w_i$, where r_i is the measurement size (a subset of the representative of cluster i), and w_i is the warm-up size.

Measurement size r_i must be smaller than budget B_i ; that is, $r_i = \min(s_i, B_i)$. Thus, we sometimes need to truncate the simulation of the cluster representative. This truncation rarely degrades accuracy, thanks to the looping behavior at the core of our region-partitioning scheme.

Because of observation 2, we preferentially allocate warm-up instructions to small samples, within the constraint of budget B_i ; that is, $w_i = B_i - r_i$. Warm-up instructions w_i are instructions preceding the representative of cluster i . Owing to our region-based partitioning approach, these instructions could reference code sections and data structures distinct from those referenced in the representative. To avoid simulating useless warm-up instructions, we use the boundary line reuse latency (BLRL) technique to determine the size of the useful warm-up interval.¹⁸ BLRL is an architecture-independent method that collects the memory addresses and branch instruction addresses used in the sampled interval and identifies the closest point in the trace before the interval where they will be all accessed. By starting the warm-up at that

Table 2. Baseline simulation model.

Component	Characteristics
Instruction cache	16 Kbytes; four-way set-associative; 32-byte blocks; 1-cycle latency
Data cache	16 Kbytes; four-way set-associative; 32-byte blocks; 1-cycle latency
L2 cache	128 Kbytes; eight-way set-associative; 64-byte blocks; 12-cycle latency
Main memory	120-cycle latency
Branch predictors	Hybrid: 8-bit gshare with 2K 2-bit predictors and an 8-Kbit bimodal predictor
OOO issue	Out-of-order issue of up to eight operations per cycle; 64-entry reorder buffer
Memory disambiguation	Load-store queue; loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating-point registers
Functional units	2 integer ALUs; 2 load-store units; 1 floating-point adder; 1 integer MULT/DIV; 1 floating-point MULT/DIV
Virtual memory	8-Kbyte pages; 30-cycle fixed TLB miss latency after earlier-issued instructions complete

point, most SRAM structures, regardless of size, are likely to be adequately warmed, meaning the first access—for example, to a cache line—will be correctly identified as a hit or a miss. However, under that constraint, the actual warm-up interval per sampled region can be very large. For example, parser requires more than 2 billion warm-up instructions for a region of only 1.8 million instructions. For that reason, we propose setting 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 used a threshold of 95 percent across all benchmarks. Still, because our budget approach introduces a size constraint on the warm-up interval, we slightly modified BLRL by limiting the warm-up size it determines to w .

Evaluation

For evaluation purposes, we used the SimpleScalar 3.0b toolset for the Alpha ISA and experimented with all 26 SPEC CPU2000 benchmarks.¹⁰ To create the regions, we used the sim-fast functional simulator. Table 2 shows the microarchitecture configuration we used for our experiments.

Figure 5 graphs the number of instructions and Figure 6 graphs accuracy for our budget approach (with budget B set to 500 million instructions) and different configurations of

SimPoint. (To provide a fair accuracy and size comparison, we set the maximum number of samples to 50 for 10-million-instruction intervals, and to 100 for 1-million-instruction intervals.) SimPoint treats sampling as an issue independent from warm-up, so we used perfect warm-up for SimPoint as do most of the articles on the subject.^{1,4,8,9} For comparison, we also made some tests of SimPoint with no warm-up. Although the accuracy of SimPoint 10M (with 10-million-instruction intervals) is barely sensitive to warm-up, SimPoint 1M becomes fairly sensitive, falling from 0.7 percent to 2.4 percent CPI error rate when we eliminate warm-up. The trend can only worsen as the sample size decreases. Therefore, although SimPoint 1M requires few instructions compared to SimPoint 10M and our budget approach with warm-up, it would actually require additional warm-up instructions to preserve its accuracy.

Our budget approach has lower accuracy but requires fewer instructions than SimPoint 10M. However, our contribution lies not so much in this instruction budget reduction, but in the fact that the user need not worry about setting the appropriate sample and warm-up sizes for a given new program; our partitioning and budgeting approach integrates all these functions. All the user needs to set or decide is the maximum simulation budget—that is, time.

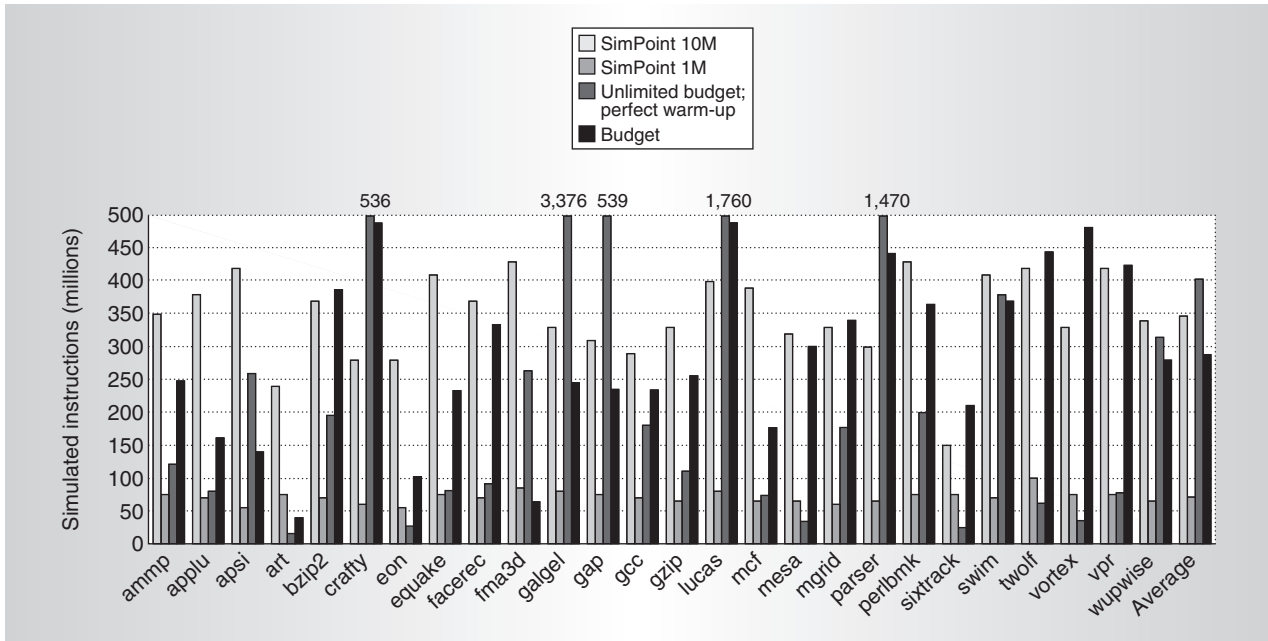


Figure 5. Number of simulated instructions for variations of SimPoint and the budget sampling technique.

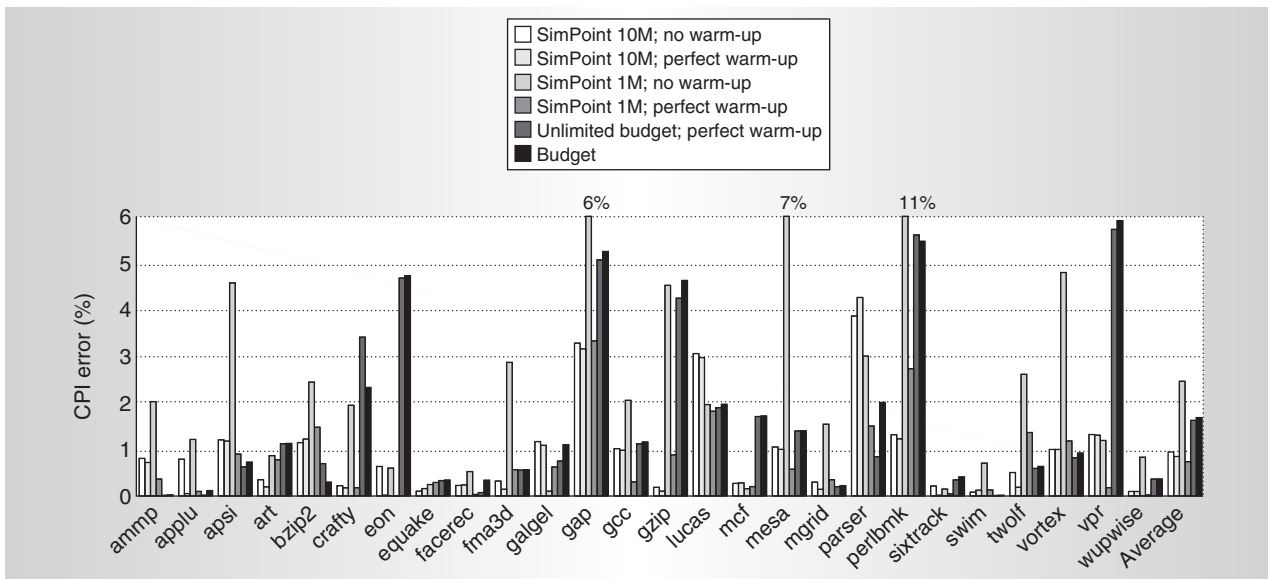


Figure 6. CPI error for variations of SimPoint and the budget sampling technique.

Still, for several cases, especially *eon* and *vpr*, our budget approach performs significantly worse than SimPoint. Some programs, such as these, have tiny but frequently recurring regions, which translate into clusters with many small intervals. Performance is more variable across small intervals; that is, it is harder to reach steady-state performance after just a few hundred thousand instructions.

This is in part due to the higher influence of start-up state on performance for such small intervals. This variability, in turn, can result in significant performance estimation error. As Table 1 shows, *eon* and *vpr* have particularly small regions on average, around 400,000 instructions. Although small regions are not necessarily synonymous with performance variability and higher error, they are a

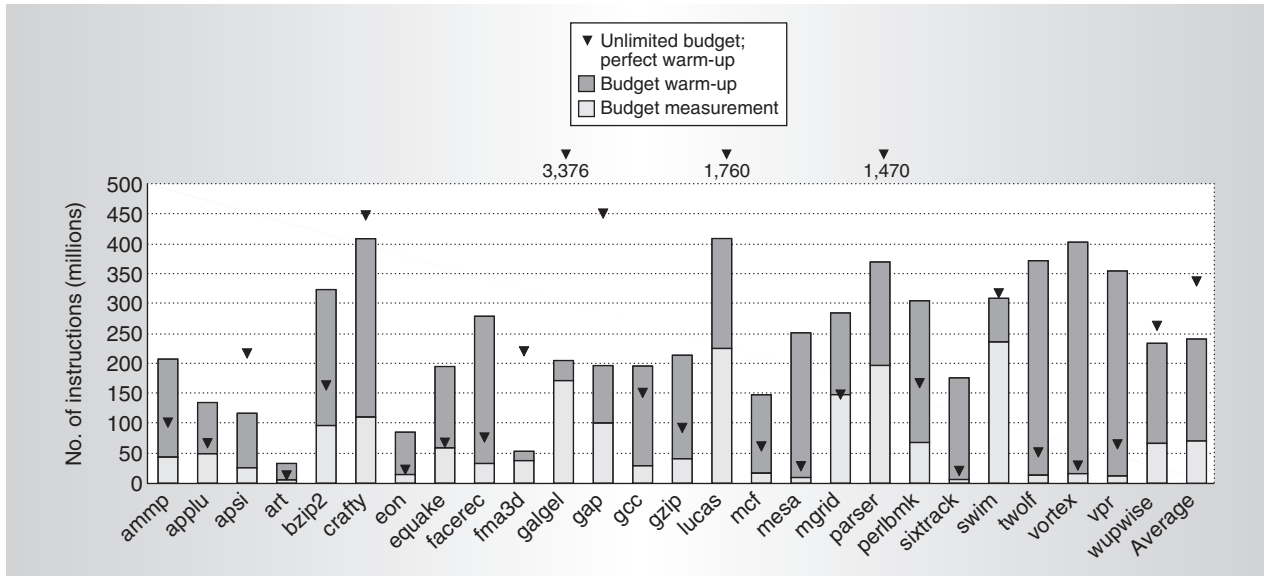


Figure 7. Budget approach distribution of measurement and warm-up instructions.

potentially aggravating factor. Still, these two codes highlight less a shortcoming than the necessity to fine-tune our heuristic: Both codes use only around 20 million instructions for sampling and 80 million overall—a small fraction of the total available budget of 500 million instructions.

A possible extension of our method could be to simulate multiple intervals within clusters where the most representative intervals are small—for example, 10 percent of the cluster budget. Not only would this average out the performance variability within such clusters, but it would also provide a means for estimating the error within such clusters. Error estimation capability would be a significant enhancement to our technique; one of the shortcomings of clustering-based techniques compared to statistical simulation techniques is that they cannot easily provide a confidence estimate of the error.⁸

Other codes, such as *gap* and *perlbnk*, also behave worse than *SimPoint 10M*. However, *SimPoint 1M* without warm-up behaves significantly worse in both cases as well. In fact, these benchmarks illustrate the difficulty of properly selecting both regions and warm-up size. They show that systematic techniques like *SimPoint* can perform well or poorly depending on how the user selects the interval size, unless the user is willing to engage in a trial-and-error process for selecting the size.

Our budget approach might not be optimal, but it does shield the user from such decisions by selecting region sizes automatically.

We also evaluated our approach with no budget limitation and no budget spent on warm-up. For the perfect warm-up with no budget limitation bars in Figures 5 and 6, none of the budget is allocated to warm-up. Wisely allocating the budget allows drastic reductions in the number of simulated instructions in several cases, with limited impact on accuracy (from 1.62 percent to 1.68 percent). In some cases, the unlimited budget requires fewer instructions than the standard budget approach; this is because the standard budget approach includes warm-up. Overall, our allocation strategies result in a total budget significantly lower than the maximum accepted budget, set at 500 million instructions in these experiments.

Figure 7 displays, for each benchmark, how our approach actually distributes its instruction budget between measurement and warm-up. The number of simulated instructions devoted to measurement is rather low—only 84 million instructions on average. This value is close to the number of instructions simulated by *SimPoint* with 1-million-instruction samples (71 million instructions). Warm-up uses most of the instruction budget, with an average of 70 percent of the total number of simulated instructions. This suggests that

warm-up and sampling should not be considered separately, especially if the goal is to develop an architecture-independent sampling method by implementing warm-up through simulation.

The rationale for our sampling approach is that some of the most recent and efficient sampling techniques have practicality shortcomings stemming from their warm-up approach. These shortcomings can make it difficult to explore specific architectural organizations or a large range of them. In addition, such sampling methods are not compatible either with current and upcoming modular simulation frameworks, where the sampling technique will be implemented in the common simulation engine. Modular simulation frameworks will require sampling techniques that are architecture-independent and transparent to the user. Our budgeting technique, which will be one of the sampling options implemented in the UniSim framework under development, meets these requirements.

MICRO

References

1. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *SIGOPS Operating Systems Rev.*, vol. 36, no. 5, May 2002, pp. 45-57.
2. R.E. Wunderlich et al., "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proc. Int'l Symp. Computer Architecture (ISCA 03)*, IEEE Press, 2003, pp. 84-97.
3. W. Liu and M.C. Huang, "Expert: Expedited Simulation Exploiting Program Behavior Repetition," *Proc. Ann. Int'l Conf. Supercomputing (ICS 04)*, ACM Press, 2004, pp. 126-135.
4. J. Lau et al., "Motivation for Variable Length Intervals and Hierarchical Phase Behavior," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS 05)*, IEEE Press, 2005, pp. 135-146.
5. T.F. Wenisch et al., "TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes," *Sigmetrics Performance Evaluation Rev.*, vol. 33, no. 1, June 2005, pp. 408-409.
6. M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Efficient Sampling Startup for Sampled Processor Simulation," *Proc. Int'l Conf. High-Performance Embedded Architectures and Compilers (HiPEAC 05)*, Springer, 2005, pp. 47-67.
7. E. Perelman et al., "Using SimPoint for Accurate and Efficient Simulation," *Sigmetrics Performance Evaluation Rev.*, vol. 31, no. 1, June 2003, pp. 318-319.
8. E. Perelman, G. Hamerly, and B. Calder, "Picking Statistically Valid and Early Simulation Points," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT 03)*, IEEE CS Press, 2003, p. 244-257.
9. J. Lau, S. Schoenmackers, and B. Calder, "Structures for Phase Classification," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS 04)*, IEEE CS Press, 2004, pp. 57-67.
10. D. Burger, T.M. Austin, and S. Bennett, *Evaluating Future Microprocessors: The SimpleScalar Tool Set*, tech. report CS-TR-1996-1308, Computer Sciences Department, Univ. Wisconsin-Madison, 1996.
11. K.C. Barr and K. Asanovic, "Branch Trace Compression for Snapshot-Based Simulation," *Proc. Int'l Symp. Performance Analysis of Systems and Software (ISPASS 06)*, IEEE Press, 2006, pp. 25-36.
12. 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," *Proc. IEEE Int'l Symp. Signal Processing and Information Technology (ISSPIT 5)*, IEEE Press, 2005, pp. 1-6.
13. J.R. Larus, "Whole Program Paths," *Proc. ACM Sigplan Conf. Programming Language Design and Implementation (PLDI 99)*, ACM Press, 1999, pp. 259-269.
14. C.G. Nevill-Manning and I.H. Witten, "Compression and Explanation Using Hierarchical Grammars," *Computer J.*, vol. 40, nos. 2/3, 1997, pp. 103-116.
15. D. Pelleg and A.W. Moore, "X-Means: Extending K-Means with Efficient Estimation of the Number of Clusters," *Proc. Int'l Conf. Machine Learning (ICML 00)*, Morgan Kaufmann, 2000, pp. 727-734.
16. D.G. Pérez, H. Berry, and O. Temam, "IDDCA: A New Clustering Approach for Sampling," *Proc. Workshop Modeling, Benchmarking and Simulation (MoBS 05)*, 2005; http://www.arctic.umn.edu/~jjyi/MoBS/2005/program/MoBS_2005_Proceedings.pdf.
17. A. Baune et al., "Dynamical Cluster Analysis

of Cortical fMRI Activation," *NeuroImage*, vol. 6, no. 5, May 1999, pp. 477-489.

18. L. Eeckhout et al., "Accurately Warmed-up Trace Samples for the Evaluation of Cache Memories," *Proc. High-Performance Computing Symp. (HPC 03)*, Soc. for Computer Simulation, 2003, pp. 267-274.

Daniel Gracia Pérez is a researcher at CEA LIST—the French Atomic Energy Commission's Lab of Applied Research on Software-Intensive Technologies—in Saclay, France. His research interests include computer architecture, simulation methodology, and simulation speed. He has master's degrees in computer science from Universitat Politècnica de Catalunya, Barcelona, Spain, and Kungliga Tekniska Hogskolan, Stockholm, Sweden, and a PhD in processor simulation from University of Paris Sud/INRIA, France.

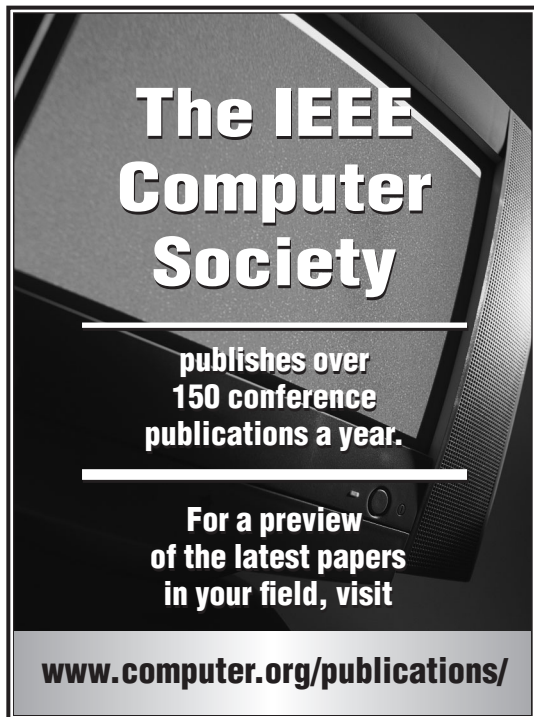
Hugues Berry is a research scientist with the Alchemy project team at INRIA, the French National Institute for Research in Computer Science and Control, in Orsay, France. His

research interests include biologically inspired approaches to computer architecture, computational neuroscience, and complex systems approaches. Berry has a PhD in biophysics from Université de Technologie de Compiègne, France.

Olivier Temam is a senior researcher at INRIA Futurs, Paris, where he heads the Alchemy group. His research interests include processor architecture and simulation, program optimization, emerging technologies and their impact on long-term architecture and programming. He has a PhD in computer science from the University of Rennes.

Direct questions and comments about this article to Olivier Temam, INRIA, Bat. N, Parc Club Orsay Université, ZAC des vignes, 4, rue Jacques Monod, 91893 Orsay Cedex, France; olivier.temam@inria.fr.

For further information on this or any other computing topic, visit our Digital Library at <http://www.computer.org/publications/dlib>.



**The IEEE
Computer
Society**

**publishes over
150 conference
publications a year.**

**For a preview
of the latest papers
in your field, visit**

www.computer.org/publications/



**JOIN A
THINK
TANK**

Looking for a community targeted to your area of expertise? IEEE Computer Society Technical Committees explore a variety of computing niches and provide forums for dialogue among peers. These groups influence our standards development and offer leading conferences in their fields.

Join a community that targets your discipline.

In our Technical Committees, you're in good company.

www.computer.org/TCsignup/