

# IDDCA: A New Clustering Approach For Sampling

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

**Abstract.** Clustering methods are machine-learning algorithms that can be used to easily select the most representative samples within a huge program trace. *k-means* is a popular clustering method for sampling. While *k-means* performs well, it has several shortcomings: (1) it depends on a random initialization, so that clustering results may vary across runs; (2) the maximal number of clusters is a user-selected parameter, but its optimal value can be benchmark/trace-dependent; (3) *k-means* is a multi-pass algorithm which may be less practical for a large number of intervals. To solve these issues, we adapted an alternative clustering method, called DCA, to the issue of sampling. Unlike *k-means*, DCA and its sampling-specific adaptation *IDDCA* do not require the user to be exposed to internal clustering parameters: it dynamically defines the number of clusters for each target program and the method parameters dynamically adapt to the target program. For an ordered input (e.g., a trace of intervals), the method is deterministic. Finally, it is an online and thus single-pass algorithm, resulting in a significant execution time gain over an existing and popular *k-means* implementation. Within the context of a variable-size sampling approach, we show that *IDDCA* can achieve an average CPI error of 1.62% over the 26 SPEC benchmarks, with a maximum error of 5.72% and an average of 403 million instructions.

## 1 Introduction

Sampling is an accurate and fast solution to increasingly long simulation times (more complex superscalar processors, multi-cores, modular simulation [12, 10], etc). There are two possible approaches for selecting sampling intervals: either (1) pick a large number of uniformly (or randomly) selected small intervals (SMARTS/TurboSMARTS [14, 13]), or (2) pick a few but large and carefully selected intervals (SimPoint [11, 5, 7, 8, 4], EXPERT [6] and our recently proposed budgeted region sampling technique BeeRS [9]). Selected sampling intervals can either have a fixed or a variable size. The sampling accuracy/size tradeoff is more easily and finely adjusted with fixed-size intervals. Variable-

size intervals [4], where the interval definition is based on program semantic, may potentially allow even more targeted, and thus accurate, sampling. However, the number of intervals is harder to control and thus potentially large, and the size of intervals can wildly vary. To date, two variable-size sampling methods have been proposed, SimPoint VLI and EXPERT. In the present article, we assume a variable-size sampling method called BeeRS [9], which relies on a simple basic block reuse distance criterion for selecting intervals. The accuracy/size target of BeeRS is to achieve an accuracy of the order of one or a few percents (sufficient for comparing architectures performance) and then to minimize the sampling size. BeeRS also particularly focuses on applicability, i.e., facilitating the use of sampling.

A key step of selecting sampling approaches is naturally how they select sample intervals. The principle is to group together similar program intervals using so-called clustering methods. Currently, SimPoint, the most popular selecting sampling approach, relies on a clustering method called *k-means*. In the present article, we present a new clustering method for sampling, within the context of BeeRS, that addresses three applicability shortcomings of *k-means* (when applied to sampling): (1) the method works by randomly selecting intervals at the start-up phase, so that several runs of the method on the same trace may not provide the same sampling intervals, and thus the same accuracy results; (2) the number of clusters is a user-selected parameter, but its optimal value can be benchmark/trace-dependent, so that inappropriately setting this parameter can degrade either simulation time or accuracy; (3) the method requires multiple passes which may be impractical for a large number of intervals.

The clustering method introduced in this article is called *IDDCA* (*Interleaved Double DCA*), and it is derived from the *Dynamical Clustering Analysis* (DCA) [1] clustering method, and adapted to sampling. We show that *IDDCA* provides consistent results across runs (no randomization phase when the input set is ordered as a set of trace intervals), it automatically determines the appropri-

ate number of samples, and it is about two orders of magnitude faster than *k-means*. The clustering technique currently used in BeeRS would fail on 3 SPEC benchmarks (it would only identify a single cluster, breeding large CPI errors). BeeRS uses a clustering technique which is inspired, but distinct, from *k-means* or its iterative variant *X-means*, and called *Unweighted X-Means*, *UXM*. Plugging *IDDCA* in BeeRS results in less than 6% CPI error on all 26 SPEC benchmarks with an average 1.62% CPI error (assuming 20M instructions warm-up before each interval, which is almost perfect warm-up).

Section 2 presents the BeeRS sampling method for partitioning the program trace into regions. Later on, all clustering techniques are only applied to this trace partitioning method. Section 3 introduces the DCA clustering algorithm, and highlight its differences with *k-means*. Finally, Section 5 presents sampling accuracy and size results and how/why *IDDCA* was derived from DCA.

## 2 Program Partitioning Into Regions

Beers (*Budgeted Region Sampling*) is a recently proposed sampling method [9] for partitioning the program trace into variable-length intervals. This method is easy to implement and tolerates irregular program control flow; other recent variable-length interval partitioning includes EXPERT [6] and SimPoint VLI [4]. In this section, we introduce the trace partitioning of BeeRS, and in the next sections we investigate the behavior of DCA and *IDDCA* on these variable-size intervals.

**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 occur-

rences 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.

**Selecting  $T$ .** The only really important parameter in this region partitioning method is  $T$ .  $T$  represents a trade-off: a too large  $T$  and the region sizes can be fairly large, degrading simulation time (in the extreme case where  $T$  yields a single region containing the whole code, the accuracy is the best possible and the simulation time is the worst possible); a too small  $T$  and the region breakdown is too fine-grained to unveil characteristic groupings of basic blocks, degrading accuracy (if  $T = 1$ , regions contain a single block and accuracy is very poor). While this trade-off seems delicate at first sight, we found the accuracy of the region partitioning method was largely, if not remarkably, tolerant to variations of  $T$ .

Still, we want to find  $T$  in a manner that is practical for the user, i.e., architecture-independent. Thus, we need to set  $T$  only once for each benchmark/data set pair (independently of the target architecture), much like SimPoint provides architecture-independent simulation points; the final recommended  $T$  values are indicated in Table 1. Let us now explain how we find these  $T$  values.

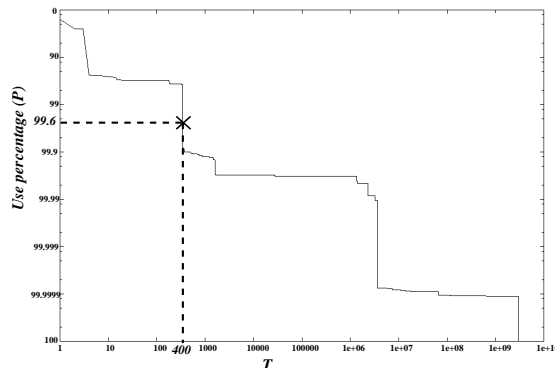


Figure 1: *Selecting  $T$  from  $P$  for `swim`.*

Since  $T$  determines what reuse distances are captured by regions, a fixed value of  $T$  can potentially miss key

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

Table 1: *Region statistics and  $T$ .*

reuses in certain programs or conversely insufficiently discriminate regions in other programs.<sup>1</sup> A more benchmark-tolerant way to capture "enough but not too many" reuses is to set  $T$  for each benchmark such that a fixed percentage  $P$  of reuse distances are captured in regions. Then, we can deduce the corresponding value of  $T$  based on the basic block reuse distance distribution, see Figure 1 for benchmark *swim*. During a training run (emulation only, no simulation, the run is architecture-independent, performed once for each benchmark/data set pair), we record the basic block reuse distance distribution, and the region partitioning for a large range of  $T$  values.

The issue now is to find the appropriate value for the percentage of reuse  $P$ .  $P$  influences accuracy and time; since time (number of simulated instructions) is an architecture-independent characteristic, we select  $P$  based on a time target (rather than an accuracy target) in order to fulfill our architecture-independence constraint. We have currently set this time target at  $\approx 200$  million instructions, but it is a user-adjustable toggle that can be increased/decreased at the benefit/expense of accuracy. Based on the time target, we now want to find  $P$ . Recall the same value of  $P$  is used across all benchmarks, so the time target will be fulfilled in average, across all bench-

<sup>1</sup>Note however that we did observe very good average accuracy/time tradeoffs for the same  $T$  value applied across all benchmarks, sustaining the above mentioned tolerance to  $T$  variations.

marks. Since we know how to find the  $T$  value for each benchmark based on  $P$  and the training run, and since we know the number of instructions (time) for each value of  $T$  thanks to the training run, we can compute the average time over all benchmarks for each  $P$  value; conversely, for a given time target, we can deduce  $P$ . Based on the training run and the average time target of 200 million instructions, we found  $P = 99.6\%$ . The corresponding values of  $T$  for each benchmark, along with other region statistics, are indicated in Table 1.

This heuristic allows to appropriately set  $T$  and to achieve a good accuracy/time tradeoff, but we do not claim it is optimal. We intend to investigate better  $T$  selection heuristics in the future.

Note that the value of  $P$  (and consequently  $T$ ) depends on the clustering method. In this article, we have estimated  $P$  using *UXM*, and we use this value throughout the article, whatever the clustering method.

***UXM and motivation for *IDDCA*.*** In order to apply SimPoint to variable-size intervals of BeeRS, the new SimPoint VLI approach should be used, which weights the intervals with their size (number of instructions) during the clustering method as described by Lau et. al. [4]. This weighting is important for accuracy because it affects the relative position of the centroid (center of mass) of a cluster of intervals with respect to all its intervals (for each cluster, the interval that is located the closest to the centroid is chosen to represent the cluster; if the centroid location is incorrectly estimated, a sub-optimal representative interval may be selected). Weighting implicitly ensures that the importance of an interval is correlated to how many instructions it contains. Since SimPoint VLI is not yet released, we instead compare *IDDCA* with *Unweighted X-means*, and we use SimPoint 2.0 as the implementation for *UXM*.

However, BeeRS aims at reducing size as much as increasing accuracy, not privileging accuracy over size at all costs. For that reason, we experimented with a method that works like *k-means* but which ignores the size of the intervals, i.e., not weighting the intervals with their size. This simple trick has the effect of avoiding that larger intervals are systematically privileged, though it may come at the expense of accuracy. This method is no longer *k-means*, hence the *UXM* name, but we found it almost achieved the intended BeeRS target accuracy with a reasonable sampling size of 160 millions instructions in average. However, the effect of not weighting the cluster can sometimes badly skew centroids locations (the unweighted centroid may be far from the more representative weighted centroid) resulting in inappropriate or wrongly chosen intervals. For 3 benchmarks, *apsi*,

`gzip` and `mcf`, this effect particularly shows since only a single cluster is identified by *UXM*, resulting in very poor accuracy for two of the benchmarks (CPI error of 26.35% for `apsi`, 3.65% for `gzip` and 93.19% for `mcf`, with a 20-million instruction warm-up before each interval).

Thus, the motivation for *IDDCA* was twofold: the shortcomings outlined in the introduction, and finding a clustering method with a reasonable accuracy/size trade-off and no inaccuracy singularity as with *UXM*.

### 3 Dynamic Cluster Analysis (DCA)

Once a program execution has been divided into intervals (called regions in BeeRS) we can cluster them on the basis of their similarities. In our case, a data point is a region, i.e., a vector of execution frequencies, one per region basic block. All the clustering results of this article are applied to the BeeRS intervals.

Briefly, the aim of clustering algorithms is to classify data points into separate clusters obeying the following rule: a given data point must be more similar to any data point belonging to the same cluster than to any data point picked in a distinct cluster. Because the number of basic blocks per region can be high, the vector dimension can be large, making clustering a fairly time-consuming task. In our case, the number of static basic blocks ranges from 1,236 for `art` to 35,202 for `gcc`. In order to reduce computation time, clustering methods usually pre-process data points by reducing the vector dimension using projection. *Random linear projection* [3, 11] is a fast and simple method to reduce the dimensions without degrading too much the information of the input data. SimPoint uses this technique to reduce the vector dimension down to 15.

#### 3.1 Algorithm

DCA is an alternative clustering method recently proposed by Baune et al. [1]. DCA dynamically defines the number of clusters and their centroids, and constantly revisits intermediate decisions. This dynamic process relies upon three different parameters:  $\Theta_{new}$ ,  $\Theta_{merge}$  and  $\Theta_{step\_factor}$ .

DCA starts with no cluster and the list of regions to cluster (called  $R$ ), and executes the following steps:

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  $\Theta_{new}$ , then create a new cluster containing the current region  $r$ .
4. If  $d$  is less or equal to  $\Theta_{new}$  then:
  - 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 or equal to  $\Theta_{merge}$  then merge the two clusters into a unique one and compute its centroid.
  - Update  $\Theta_{new}$  and  $\Theta_{merge}$  thresholds so as to make cluster creation and merger more difficult. For that purpose, increase  $\Theta_{new}$  and decrease  $\Theta_{merge}$  as follows:  $\Theta_{new} = \Theta_{new} / \Theta_{step\_factor}$  and  $\Theta_{merge} = \Theta_{merge} \times \Theta_{step\_factor}$  (for  $\Theta_{step\_factor} < 1$ ).
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 DCA has created a set of clusters. The sampled points are the closest regions to the clusters centroids.

Intuitively,  $\Theta_{new}$ ,  $\Theta_{merge}$  and  $\Theta_{step\_factor}$  control the creation and merging of clusters. Their initial values are benchmark specific. We first compute the centroid of all the regions (taken as a whole);  $\Theta_{new}$  and  $\Theta_{merge}$  are then initialized to 10% of the distance between this global centroid and the farthest region.  $\Theta_{step\_factor}$  determines the rate at which the probabilities of creating or merging clusters changes. We found that  $\Theta_{step\_factor}$  depends on the number of data points to cluster, but that the method was robust enough to tolerate the same value across all benchmarks. We empirically found  $\Theta_{step\_factor} = 1 - 10^{-5}$  to be appropriate for the SPEC size range.

#### 3.2 DCA versus *k-means* and *UXM*

In this section, we compare the algorithmic assets of DCA over *k-means* and its iterative variant *UXM*.

**Exposing the user to internal parameters.** In *UXM*, the maximum number of clusters is defined by the *max\_k* parameter, but this parameter is arbitrarily set by the user. While many codes perform well (good accuracy) with the same *max\_k* (or  $k$ ) values, some codes require high *max\_k*, or conversely work well with a low *max\_k* ( $k$ ) (and thus require few sampling intervals). Table 2 shows the number

SPEC	max_k		
	10	50	100
ampp	10	45	100
applu	10	10	10
apsi	1	1	1
art	10	10	10
bzip2	10	32	100
crafty	1	17	100
eon	10	31	100
equake	10	17	17
facerec	10	28	28
fma3d	1	20	100
galgel	10	10	10
gap	10	32	100
gcc	4	4	4

SPEC	max_k		
	10	50	100
gzip	1	1	1
lucas	1	13	13
mcf	1	1	1
mesa	10	19	19
mgrid	10	16	16
parser	10	12	100
perlbmk	1	41	100
sixtrack	10	37	100
swim	10	24	24
twolf	10	50	100
vortex	10	49	100
vpr	10	29	100
wupwise	10	16	16
Average	7	22	53

Table 2: Number of clusters obtained with different  $max_k$  values using UXM for BeeRS intervals.

of clusters for different values of  $max_k$ . Obviously some benchmarks require a significantly higher number of clusters than others. SimPoint 2.0 uses a back-off heuristic to systematize the setting of  $max_k$ : if the number of clusters found is equal to  $max_k$ ,  $max_k$  is doubled and the clustering run again.

The DCA algorithm also has internal parameters ( $\Theta_{new}$ ,  $\Theta_{merge}$  and  $\Theta_{step\_factor}$ ), but the initial values of these parameters have less impact on the algorithm behavior because their values are dynamically adjusted during the algorithm execution. In practice, for none of the experiments we had to tailor these parameters to the benchmarks.

**Clustering speed.** Even though the complexity of UXM and IDCCA are similar, the implementation of IDCCA is 130 times faster than a widely used implementation of  $k$ -means, i.e., the SimPoint 2.0 implementation [8].<sup>2</sup> For instance, clustering the BeeRS intervals for one benchmark with all of the same parameters in `runsimpoint` script except  $max_k = 100$  requires 21 hours in average on a modern PC, and up to two days (`crafty`). In average, IDCCA requires 9 minutes for the same interval list, and 44 minutes on `crafty`.

The SimPoint group has apparently developed a new version of its clustering method [4], still based on  $k$ -means, and adapted to variable-size intervals, but it has not been released yet. This new version also proposes to speed up the clustering process on very large inputs (very large numbers of intervals) by sub-sampling the set of intervals to cluster, and run  $k$ -means on only this sample. While this technique can greatly improve the clustering speed, it can also degrade the clustering quality. Note that this technique can also be applied to DCA, so it shifts rather than bridges the performance gap.

<sup>2</sup>Recall that UXM is based on SimPoint 2.0.

SimPoint has also sped up the iterative clustering process by limiting the number of iterations of the  $k$ -means algorithm, even if it has not converged (100 iterations by default). We have found no case on the 26 Spec benchmarks where this fixed  $max_k=100$  threshold would not be sufficient, but it is also difficult to ensure that a fixed parameter will be compatible with any benchmark.

While  $k$ -means must compute multiple times the distance between the cluster centroids and the regions, DCA only needs to compute this distance once for each region. We can take advantage of this higher speed to improve DCA accuracy, when applied to sampling, by using a large dimension after the *random linear projection* (100).

**Clustering stability.** The clustering quality achieved by the *random* assignment of the initial centroids. Thus, clustering quality can vary from one run of the clustering method to another; therefore, it may be hard for researchers to replicate the experiments of other researchers. Note that, to address this  $k$ -means issue, SimPoint 2.0 uses a modified initialization method, called *furthest-first*, that allows to partly reduce this source of variability. This initialization technique randomly selects a centroid from the region space, and then recursively selects new centroids from the region space such that their distance to already chosen centroids is maximized. Additionally, SimPoint proposes to execute  $k$ -means multiple times (5 by default) to maximize the likelihood that the best clustering is obtained.

Section 5 presents the simulation accuracy results for DCA.

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

Table 3: Baseline simulation model.

## 4 Methodology

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 3 shows the microarchitecture configuration used for our experiments.

**Warm-up.** Even though BeeRS [9] proposes a budget-aware warm-up technique, the issue of warm-up is out of the scope of this study. Therefore, to minimize the impact of warm-up, we add a significant warm-up interval of 20 millions instructions before each region, corresponding to a total warm-up size of several billions instructions per benchmark (almost perfect warm-up). However, we have demonstrated that the BeeRS warm-up technique can achieve a similar or better accuracy with a warm-up size of about 100 millions instructions per benchmark.

## 5 Adapting DCA to sampling

### 5.1 DCA

Table 4 and Figure 2 respectively show the number of clusters and the sampling size for DCA. While DCA created a reasonable quantity of clusters for most of the benchmarks, it generated more than 100 of them for `bzip2`, `galgel`, `gcc` and `parser`, resulting in very large sampling sizes. For instance, DCA detected  $\approx 500$  clusters for `parser`, which is the origin of the large number of instructions simulated for this benchmark.

More subtle clustering issues also appear in Table 4. For instance, for `galgel` and `lucas`, DCA produced several clusters that mainly contained very large regions (of more than 400 million instructions). Simulation of these clusters greatly increased the number of instructions required for these benchmarks.

Figure 3 shows the accuracy results for DCA. Even though DCA exhibited no CPI error as strong as `UXM`, it still fails significantly for 3 codes: `gap`, `perlbnk`, and in a lesser way, `quake`. In the next section, we further analyze the behavior of DCA on these benchmarks, and we present enhanced DCA algorithms.

### 5.2 Interleaved DCA

As explained in Section 3.1, when processing the first regions of a program, DCA easily creates and merges clusters, but as more regions are processed, the probability to create new clusters decreases, keeping the number of clusters stable. This method works well when the overall variability of the regions is experienced from the beginning of

SPEC	DCA	IDCA	IDDCA	SPEC	DCA	IDCA	IDDCA
ampp	45	49	49	gzip	59	167	167
applu	33	37	37	lucas	60	56	56
apsi	50	44	44	mcf	36	54	54
art	46	42	42	mesa	8	16	16
bzip2	145	318	318	mgrid	36	32	32
crafty	20	10	527	parser	495	507	507
eon	20	92	92	perlbnk	33	27	129
quake	14	17	17	sixtrack	46	46	46
facerec	24	22	22	swim	53	54	54
fma3d	70	73	73	twolf	22	21	28
galgel	138	140	140	vortex	29	31	31
gap	16	92	92	vpr	91	155	155
gcc	318	323	323	wupwise	18	16	16
				Average	74	94	118

Table 4: Number of clusters obtained using DCA, IDCA and IDDCA.

the program. But whenever distinct regions appear only late in the program execution, creating a new cluster for these regions becomes more difficult.

This problem is mainly due to the fact that DCA clusters the regions in the order of their appearance along the program execution. To overcome it, a simple method consists in picking the regions to cluster at regular intervals along the execution trace. We have used this option and set the interval to one tenth of the total number of regions. Let  $N$  be this number. Hence, we first cluster the first region, then region number  $N + 1$ , followed by region  $2N + 1, \dots$ , then region 2, then region  $N + 2$ , region  $2N + 2, \dots$ . We call this variation of DCA *Interleaved DCA* (IDCA). Figure 4 shows the clustering generated by IDCA for `quake`, and compares it to the clustering generated by DCA. Clearly, IDCA created a new cluster at the end of the program that DCA did not detect. The presence of this cluster at the end of the program is further confirmed by a SimPoint clustering using fixed-size 10-million instructions intervals.

Figure 3 shows the accuracy results with IDCA, and compares them to DCA. We can see that the CPI errors for `quake` and `gap` are significantly reduced by IDCA (respectively from 9.45% to 0.33% and from 18.10% to 5.07%), thus validating our approach. But as a side effect, IDCA increased the number of instructions to simulate due to the larger number of clusters created, see Figure 2 and Table 4. We are currently working on methods for reducing the number of simulated instructions in DCA and IDCA, by biasing the cluster representative to the smallest regions.

### 5.3 Interleaved Double-DCA

Still, the accuracy of `perlbnk` remains poor, see Figure 3. An analysis of the clusters shows that there are

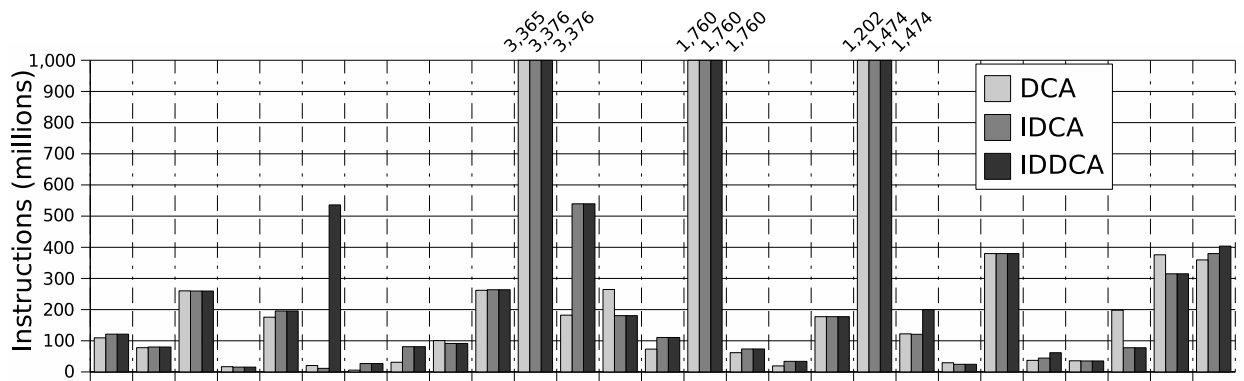


Figure 2: Simulated instructions with DCA, IDCA and IDDCA.

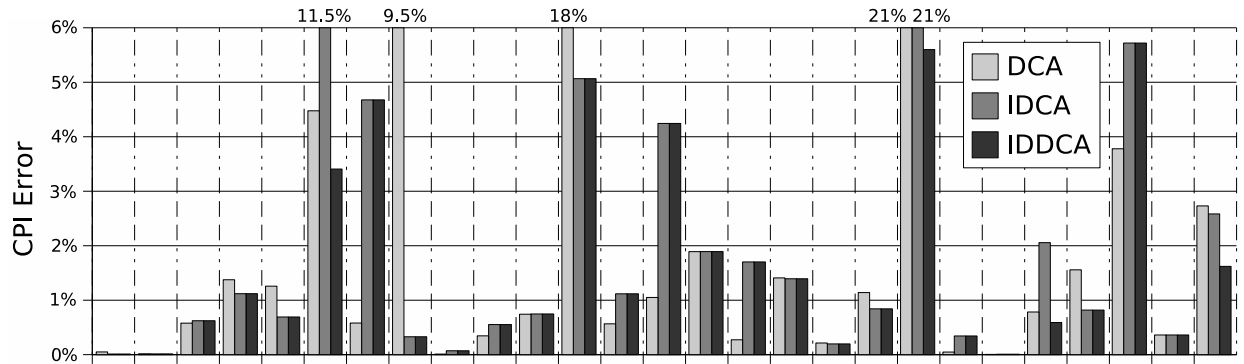


Figure 3: Simulation accuracy with DCA, IDCA and IDDCA.

“giant” clusters, i.e., clusters containing more than 90% of the program regions. `crafty` and `twolf` clustering also contain such “giant” clusters, albeit their detrimental influence on simulation accuracy was less pronounced than for `perlbnk`.

Intuitively, a first solution would consist in decreasing the initial value of  $\Theta_{new}$ , but this solution would remove much of the practicality of DCA by forcing the user to carefully set a parameter. An alternative solution, called *Interleaved Double-DCA (IDDCA)*, consists in creating an initial clustering using IDCA, then checking if one of the clusters contains more than 90% of the regions. If such a “giant” cluster is present, *IDDCA* performs a second IDCA clustering, restricted to these “giant” clusters.

Table 4 and Figure 2 show the number of clusters and simulated instructions generated by *IDDCA*. Note that the *IDDCA* double clustering was necessary only

for `crafty`, `perlbnk` and `twolf`. As expected, the number of clusters and instructions increased for these three benchmarks, but while the number of clusters for `crafty` and `perlbnk` rose from 10 to 527 and from 27 to 129 respectively, for `twolf` the number of clusters varied much more moderately, increasing from 21 to 28.

Figure 3 shows *IDDCA* clustering accuracy results, and compares them to DCA and IDCA. *IDDCA* reduced the CPI error of the three previously mentioned benchmarks (from 11.5% to 3.4% for `crafty`, from 21% to 5.6% for `perlbnk` and from 2% to 0.6% for `twolf`) and obtained the lowest average CPI error with 1.62%. Hence, *IDDCA* allowed to efficiently cluster every of the 26 SPEC benchmark studied.

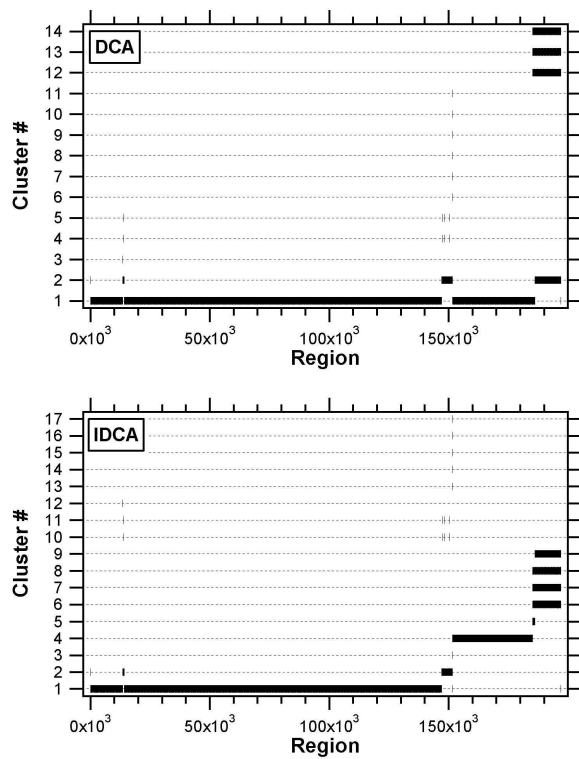


Figure 4: *equake clustering*.

#### 5.4 Unweighted vs. Weighted *IDDCA*

In all the above DCA variants, the intervals are not weighted by their size during the clustering. As for *UXM*, not weighting the clusters is a choice targeted at reducing sampling size rather than privileging accuracy. The *IDDCA* results show that an effort is still needed on size rather than accuracy, so the choice falls on the right side of the tradeoff.

Still, in order to investigate the effect of not weighting the clusters, we have run *IDDCA* clustering with weighted clusters. As expected, the total sampling size increases, and more surprisingly, significantly (34%), see Figure 5. Even more surprising, the accuracy is lower with weighted clusters than with unweighted clusters: 2.00% CPI error for weighted clusters versus 1.62% for unweighted clusters, see Figure 6. The stiff sampling size increase validates the approach of not weighting the interval size during clustering. The performance of unweighted clustering with respect to weighted clustering, both in terms of size and accuracy, suggests that there are many intervals of very different sizes with similar behavior. Thus, not weighting the intervals will still allow to pick an interval representative of the overall behavior, and

it will provide an opportunity to pick the cluster representative in an area with smaller intervals.

## 6 Conclusions and Future Work

In this article, we present *IDDCA*, a clustering method adapted to sampling and based on DCA. *IDDCA* improves DCA by reducing the effect of heterogeneous regions distributions and “giant” clusters. *IDDCA* achieves a CPI error of only 1.62% with 400 million instructions per benchmark in average. This rather large sampling size is especially due to 3 codes with samples that exceed 400 million instructions (up to 1 billion instructions sample in the case of *lucas*). We are now working on controlling such sampling size excesses within *IDDCA*. We are also investigating methods for biasing the selection of the representative regions in order to reduce the total number of simulation samples, while preserving accuracy. Furthermore, we are working on the integration of *IDDCA* into *BeeRS*, in place of *UXM*.

## 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] Sanjoy Dasgupta. Experiments with random projection. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 143–151, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- [4] 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.
- [5] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for Phase Classification. *ISPASS '04: IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.
- [6] 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.
- [7] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.

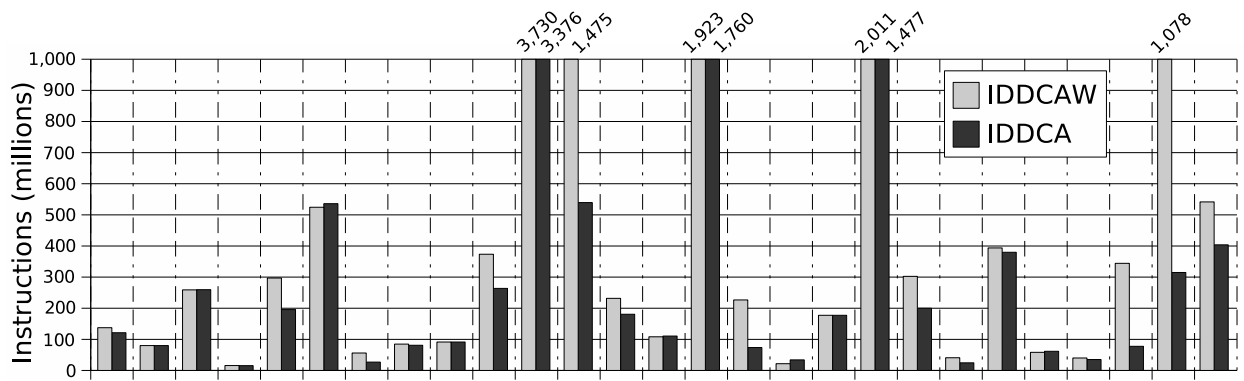


Figure 5: Simulated instructions with IDDCA and weighted IDDCA (IDDCAW).

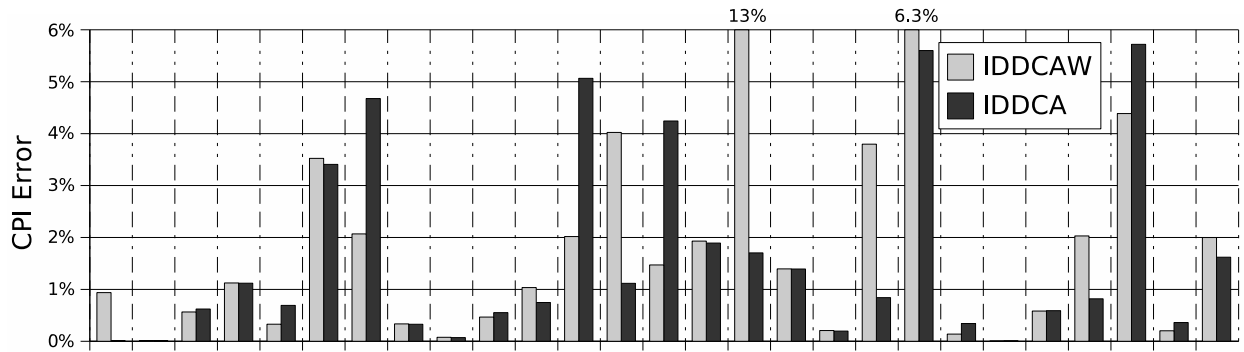


Figure 6: Simulation accuracy with IDDCA and weighted IDDCA (IDDCAW).

- [8] 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.
- [9] Daniel Gracia Perez, Hugues Berry, and Olivier Temam. Budgeted Region Sampling (BeeRS): Wisely Allocating Simulated Instruction for a Better Accuracy/Speed/Applicability Tradeoff. *Submitted for publication*, 2005.
- [10] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. In *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, pages 43–54. IEEE Computer Society, 2004.
- [11] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, 2002.
- [12] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural Exploration with Liberty. In *the 34th Annual International Symposium on Microarchitecture*, Austin, Texas, USA., December 2001.
- [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.