In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT),

September 2003.

Picking Statistically Valid and Early Simulation Points

Erez Perelman Greg Hamerly Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{eperelma,ghamerly,calder}@cs.ucsd.edu

Abstract

Modern architecture research relies heavily on detailed pipeline simulation. Simulating the full execution of an industry standard benchmark can take weeks to months to complete. To address this issue we have recently proposed using Simulation Points (found by only examining basic block execution frequency profiles) to increase the efficiency and accuracy of simulation. Simulation points are a small set of execution samples that when combined represent the complete execution of the program.

In this paper we present a statistically driven algorithm for forming clusters from which simulation points are chosen, and examine algorithms for picking simulation points earlier in a program's execution - in order to significantly reduce fast-forwarding time during simulation. In addition, we show that simulation points can be used independent of the underlying architecture. The points are generated once for a program/input pair by only examining the code executed. We show the points accurately track hardware metrics (e.g., performance and cache miss rates) between different architecture configurations. They can therefore be used across different architecture configurations to allow a designer to make accurate trade-off decisions between different configurations.

1 Introduction

Understanding the cycle level behavior of a processor running an application is crucial to modern computer architecture research. To gain this understanding, detailed cycle level simulators are typically employed. Unfortunately, this level of detail comes at the cost of speed, and simulating the full execution of an industry standard benchmark on even the fastest simulator can take weeks to months to complete. This fact has not gone unnoticed, and several techniques have been developed aimed at reducing simulation time.

For architecture research it is necessary to take one instance of a program with a given input, and simulate its performance over many different configurations for an architecture feature, searching the design space for Pareto optimal points in terms of performance, area and power. The same program binary with the input may be run hundreds or thousands of times to examine how, for example, the effectiveness of a given architecture changes with its cache size. To address these issues, we created a tool called SimPoint [18, 19]. SimPoint intelligently chooses a set of samples called *Simulation Points* to perform targeted program analysis. These provide an accurate picture of the complete execution of the program.

1.1 SimPoint Overview

To use the SimPoint analysis for simulation, the user has to specify a desired sample size. This represents the number of detailed instructions the user wants to simulate for each sample. We use a sample size of 1 million instructions for the majority of the results in this paper, but larger or smaller sample sizes can be used. For a program/input pair, the length of the program's execution is partitioned into N consecutive intervals, where each interval is equal to the length of the sample size (in terms of executed instructions). A code profile is then gathered for each interval. The profile measures the frequency of the code being executed during each interval. For each interval a *Basic Block Vector* (BBV) is created [18]. The BBV keeps track of the number of instructions executed for each static basic block for that interval. After it is generated, the BBV is normalized with the total number of instructions executed in the interval. This allows us to perform a vector difference between two intervals to see how similar they are to one another, in terms of the code they executed.

For the basic block vectors, the number of dimensions is the number of executed basic blocks in the program, which ranges from 2,756 to 102,038 for the SPEC 2000 programs, and can grow into the millions for very large programs. Therefore, we use *random linear projection* [5] to create new low-dimensional vectors and use these projected basic block vectors when comparing the similarity of two intervals. We found that projecting the data down to 15 dimensions is sufficient to still differentiate the different phases of execution [19].

The interval profiles (projected BBVs) are then used to determine the similarity of each interval with other intervals of execution. Machine learning clustering algorithms are used to group together (into clusters) intervals of execution that execute the same code blocks with the same frequency. To cluster, the difference between two interval BBVs is calculated to determine this similarity. Each cluster represents a group of potentially discontinuous execution intervals that were similar to each other. Once all the intervals for a program/input pair have been put into their clusters, we pick a single point (interval) from each cluster (appropriately weighted) to serve as that cluster's representative. These are called the *Simulation Points*. The set of representative simulation points are where detailed simulation and program analysis should be performed. Only these points are simulated, and the overall simulation metrics are calculated by weighing each simulation point by the percent of execution the

cluster it came from represents. These results are then combined to arrive at overall metrics that represent the complete execution of the program. Simulating only these points provides an accurate and efficient representation of the complete execution of the program.

The key to our approach is that for a given binary and input, the simulation points only need to be chosen once, and they are chosen by running a fast code profiler without considering any of the underlying architecture features. In our prior approach, we select these points using a method that is completely independent of any particular architecture configuration. The simulation points are selected using a metric that is only based on the code that is executed over time for a program/input pair. Once SimPoint has chosen the simulation points, they can be used for the hundreds or thousands of independent simulations to examine trade-offs in architecture design space exploration.

An important step in the SimPoint algorithm is clustering. The goal of clustering is to divide a set of points into groups such that points within each group are similar to one another (by some metric, often distance), and points in different groups are different from one another. To perform the clustering we use the popular k-means [14] algorithm, since it is a very fast and simple algorithm that yields good results. SimPoint uses k-means to group all of the code profile intervals with similar behavior (based only on the code profiles) into k separate clusters. The issue is how to choose the value of k (the number of desired clusters). Since the clustering algorithm is very fast, our approach is to examine many different clusterings (many different values of k), and then use a selection criteria for choosing the best k clustering to use. Since clustering groups points together into a cluster that are similar to each other, the larger the k, the more homogeneous the behavior becomes within each cluster. Taking this to the extreme, if every interval of execution is given its very own cluster, then every cluster will have perfect homogeneous behavior. Our goal is to choose a clustering with the *minimum number of clusters* k where each cluster has reached a certain level of homogeneous behavior.

SimPoint allows a user to specify the maximum number of clusters to be considered. For simulation, a user may want to set this to the maximum number of intervals the user is willing to perform detailed simulation on for each program, or some other criteria. In a prior study [19], we set the max K to be 10, since we used an interval size of 100 million, and this would guarantee that SimPoint would search for solutions that resulted between 100 million to 1 billion instructions for detailed pipeline simulation. In this paper, when using an interval size of 1 million we set max K to be 300, so anywhere between 1 million and 300 million detailed instructions will be simulated.

After SimPoint performs a clustering for each k up to the specified maximum K, a selection criteria needs to be used to choose the minimum k where each cluster has reached a certain level of homogeneous behavior. In [19], we used the *Bayesian Information Criterion* (BIC) score [9, 17]. This rates each cluster with a likelihood of producing a spherical Gaussian distribution along with a penalty based on the number of clusters being used. Clusterings that are well formed will have higher BIC scores. A heuristic was then used to choose a small clustering with high BIC. The heuristic takes the smallest k such that its BIC score is at least 80% as good as the best BIC score.

After picking a clustering, the final step of the algorithm is to pick a representative point from each cluster. In order to pick this representative, we choose for each cluster the actual interval that is closest to the centroid (or center) of the cluster (as measured by Euclidean distance). The *Centroid* of a cluster is analogous to the center of mass of the cluster in basic block space. Because of this, the centroid is the point that is closest to all the other points in the cluster. By choosing the interval closest to the centroid as a representative for that cluster we ensure that we are picking an interval of execution whose code usage represents the average behavior of all the intervals of the cluster.

Choosing a point close to the centroid is important, especially when the max K is set small when clustering a program. If a user wants to limit the max K to be small, for example in order to reduce simulation time, SimPoint can still provide accurate results since a point close to the centroid (the average code usage of the cluster) is picked to represent the cluster. For example, if you limit a clustering of `gcc` to be at max 300 intervals for an interval size of 1 million instructions, then we have found the clusters formed to have very similar IPC. But, if you limit `gcc` to have only 10 clusters using an interval size of 100 million, then some of the clusters will have varying IPC. In this case, since a point close to the centroid is chosen, this point represents the average code behavior of the interval, and we have found that this point also tends have the average performance (e.g., IPC) of that cluster. This allows SimPoint to still provide accurate results even when a cluster does not have homogeneous behavior, which occurs when max K is restricted to reduce simulation time.

1.2 Improvements to SimPoint

In this paper we present an algorithm for choosing the number of clusters to use by augmenting the BIC with confidence information to create a *Variance SimPoint* algorithm. We use statistical analysis to guide the picking of a clustering to provide a given level of confidence and probabilistic error bound. In addition, this same analysis can be used as a post step if desired to provide a confidence and error bound for a clustering derived using the original SimPoint BIC algorithm.

To choose a clustering (k) using our *Variance SimPoint* algorithm, we run the program/input pair once, sampling the performance over the program's execution. We then calculate the BIC as before. This is used to set a lower bound on k. We then compute the variance of each cluster, and choose the smallest k, but no smaller than the k chosen by the BIC criteria. This guides the picking of k through a user specified level of confidence and probabilistic error bound. The simulation point within each cluster is chosen as before, and this set of simulation points can then be used to guide architecture simulations across many different architecture configurations.

Note, this confidence and probabilistic error bound *only* applies to the architecture configuration used to perform the sampling. Even so, the particular confidence and probabilistic error bound are used to only choose a given clustering to augment the

clusters chosen by the BIC. This ensures that we still are choosing a clustering first by only examining the code being executed (by using the BIC to set the minimum k). The simulation points can be generated once for a program/input pair, and then used across different architecture configurations for accurate architecture research.

The absolute error of a program/input run on one hardware configuration is not as important as tracking the change in metrics across different architecture configurations. We show that our simulation points accurately track the relative change in hardware metrics across different architecture configurations. This shows that the simulations points can be used across different architecture configurations to allow a designer to accurately trade-off design points. To show this we examine many different architecture configurations varying the first and second level cache sizes, access times, and associativities.

The final contribution of this paper deals with picking earlier simulation points to decrease simulation time. For a given clustering, our prior algorithms focus on picking the most representative point from a cluster, since we assumed a simulation environment with check-pointing. Check-pointing is the process of storing the state of a simulator so that simulation can continue from that point at a later time. If the start of all of the simulation points are check-pointed, then the full program can be simulated in parallel very quickly simulating each checkpoint independently. In practice, not all simulation environments have the support for check-pointing, and instead the simulator must fast-forward (perform functional emulation) between the simulation points. This can take a significant amount of time, especially if the simulation point is at the end of execution. Therefore, we created algorithms for picking simulation points that are earlier in the program to significantly reduce the fast-forward time needed for these simulation environments.

The rest of the paper is laid out as follows. We begin with a description of prior research on simulation infrastructure in Section 2. We then briefly describe the methods used to perform this research in Section 3. In Section 4, we describe using confidence and error to guide the picking of k, and we provide a comparison to statistical sampling [4, 22]. Section 5 details our modified algorithm for finding early simulation points that are more fast-forwarding friendly. Section 6 provides empirical results showing that the simulation points accurately track performance changes across different architecture configurations. Section 7 summarizes the paper.

2 Related Work

Modern architecture research relies heavily on detailed pipeline simulation. Simulating the full execution of an industry standard benchmark can take weeks to months to complete. SimpleScalar [1], one of the faster cycle-level simulators, can simulate around 400 million instructions per hour. Unfortunately many of the new SPEC 2000 programs execute for 300 billion instructions or more. At 400 million instructions per hour this will take approximately 1 month of CPU time. This problem has motivated several researchers to develop ways of reducing simulation time while remaining true to the full simulation. In this section we provide a short summary of research related to efficient simulation.

2.1 Fast-forwarding and Checkpointing

Historically, researchers have simulated from the start of the application, but this usually does not represent the majority of the program's behavior because the code that is executing is often times performing simple chores like setting up and clearing the data structures to be used. Recently, researchers have started to *fast-forward* to a given point in execution, and then start their simulation from there, ideally skipping over the initialization code to an area of code representative of the whole. While fast-forwarding, the simulator simply acts like an emulator. When the fast-forward point has been reached, the simulator switches to full cycle-level simulation.

An alternative to fast-forwarding is to use checkpointing to start the simulation of a program at a specific point. With checkpointing, code is executed to a given point in the program and the state is saved, or checkpointed, so that other simulation runs can start there. This provides the ability to simulate all of the samples in parallel obtaining very fast results using distributed computing [13]. Checkpointing, while powerful, does come at a cost. The checkpoints themselves can be quite large since the contents of main memory needs to be saved. In addition, a checkpoint can contain saved architectural state to avoid the need to warm-up structures. If one requires many samples to guide simulation, the number of checkpoints can be a prohibitive cost. In this case, fast-forwarding is a likely alternative.

2.2 Warm-Up

After fast-forwarding, the state of caches, branch predictors, and other structures that rely on past run-time information will not be in the state that they would be in if detailed simulation had been performed. At this time the simulator is said to be cold or stale. To get accurate results for these structures, there are several options that have been proposed.

One way to look at warm-up is to try to effectively remove the cold-start bias that occurs at the start of each sample. Wood *et al.* [21] examined estimating the miss ratios for caches at the start of a sample for unknown (cold) references. They found that the miss rate of unknown references correlates to the percentage of time that a block is dead. In [10], they extend this work to perform a comparison between trace sampling techniques and examine several techniques to mitigate the cold-start bias. These range from using half of the trace to initialize the cache, to using the state of the cache from the prior trace interval (stale-state), to assuming a fraction of the initial cold-start references hit in the cache.

Conte *et al.* [3] examined a single pass approach to warming up cache structures by using part of the sample to warm-up the structure before starting to collect the cache statistics. In addition, they examined using the stale-state from the prior sample as the starting state for the next sample for architecture structures like branch prediction [4].

Recently, Haskins and Skadron [6, 8, 7] have examined finding the minimum distance to fast-forward before a simulation

point based upon the working set size of the structure to be warmed up. They use their reuse analysis to accurately determine how far ahead to warm-up different structures (e.g., branch predictors, and caches) before starting detailed simulation. Their warm-up reuse analysis fits well with our simulation points. Once you have a set of simulation points for a program/input pair, you can perform their reuse analysis to figure out how much before the simulation point structure simulation needs to start to reduce warm-up effects.

2.3 Statistical Sampling

Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. Laha *et al.* [12] introduced the use of random sampling to evaluate cache memory performance. They compared the accuracy of the sampled mean and they examined the distribution of random sampling showing that it matched the distribution of the real trace.

Conte *et al.* [4], was the first to apply statistical sampling to processor simulation calculating confidence and probabilistic error bounds for performance. They show how to calculate these bounds from a single sampling simulation run. Wunderlich *et al.* [22] extends this work using statistical sampling to focus on how to achieve accurate results for very small sample sizes on the order of 1000 instructions, and use statistical analysis from one run to determine how many samples are needed to achieve a desired error bound for a level of confidence for future runs.

In [22], they use statistical sampling to provide low error bounds $\pm$3% on average, with a high confidence 99.7% on average. They report that their approach provides an average 0.6% error in CPI, and when they used the simulation points from our website they achieved and average error rate of 3.7%. This increase in accuracy comes at a cost in the time it takes to perform simulations. On average they showed that it took 5 hours to perform a simulation run using SMARTS [22], whereas they found using our original SimPoint algorithm and the simulation points from [19] took 2.8 hours on average. Our prior results [19] used 100 million instruction sample sizes and used $K = 10$ as the maximum number of clusters. One of the contributions the work in our paper is to reduce the fast-forward time by 3.6 times over the original SimPoint algorithm in [19] by focusing on picking early simulation points, which are still close to the centroid.

The statistical sampling results in [22] requires 1,000s to 10,000s of samples to be taken over the complete execution of the program. Since this many random and small samples have to be taken over the whole program, it is harder to benefit from checkpointing or the warmup techniques in [8]. To address this, SMARTS keeps the state of certain architecture components warm (e.g., caches) during fast-forwarding. To make this feasible, their infrastructure focuses on providing fast functional simulation and efficient structure simulation during fast-forwarding. Keeping the cache structures warm will increases the time it takes to perform fast-forwarding. In comparison, SimPoint focuses on a small number of simulation points, so it can better benefit from checkpointing and pre-calculating warmup distances. Even so, keeping functional state warm during fast-forwarding can be an approach for dealing with warm-up for Early SimPoint, since it significantly reduces the fast-forward time.

2.4 Reduced Input Set

One approach for reducing the simulation time is to use the training or test inputs from the SPEC benchmark suite. For many of the benchmarks, these inputs are either (1) still too long to fully simulate, or (2) too short and place too much emphasis on the startup and shutdown parts of the program's execution, or (3) inaccurately estimate behavior.

KleinOsowski *et al.* [11], have developed a technique where they manually reduce the input sets of programs. The input sets were developed using a range of approaches from truncation of the input files to modification of source code to reduce the number of times frequent loops were traversed. For these input sets they develop, they make sure that they have similar results in terms of IPC, cache, and instruction mix.

2.5 Statistical Simulation

Another technique to improve simulation time is to use statistical simulation [16]. Using statistical simulation, the application is run once and a synthetic trace is generated that attempts to capture the whole program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. Nussbaum and Smith [15] also examined generating synthetic traces and using these for simulation and was proposed for fast design space exploration.

3 Methodology

In performing this research we made use of three different tools, ATOM [20], SimpleScalar3.0c [1], and SimPoint [19]. These tools are all designed or configured to work with the Alpha AXP ISA or are platform agnostic. ATOM, a binary modification tool, is used to quickly gather profiling information in the form of basic block vectors which can be fed as input to the SimPoint tool. The SimPoint tool then takes these vectors and performs k-means clustering analysis, BIC scoring, and determines the set of simulation points to use for that program/input pair. For all the results in this paper we use perfect warm-up. This is to remove the cold-start bias, and to focus our study on only the error differences between the simulation points picked by SimPoint and the samples chosen using statistical sampling.

SimpleScalar is used to collect the full detailed results, collect random samples, and to validate the phase behavior we found when clustering our basic block profiles. The baseline microarchitecture model we simulated is detailed in Table 1. We simulate an aggressive 8-way dynamically scheduled microprocessor with a two level cache design. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. We modified SimpleScalar to enable fast-forwarding between an arbitrary number of non-contiguous detailed simulation intervals, in one serial run. This modification is necessary for random sampling, since it involves hundreds of non-contiguous interval simulations from each program.

I Cache	8k 2-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Pred	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mem Disam	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Mem	8k byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

We analyzed and simulated the SPEC 2000 benchmarks compiled for the Alpha ISA for multiple inputs. We provide results for 43 program/input combinations. For some of the graphs we focus in on a subset of 10 program/input combinations that represent some of the more complex phase behavior found in the SPEC benchmark suite. In these graphs, the `avg-rest` results show the average for all the 33 program/input combinations not individually shown, and then `avg-all` show the average for all 43 program/input combinations. The binaries we used for this study and how they were compiled can be found at `http://www.simplescalar.com/`.

4 Picking Simulation Points Using Statistical Analysis

In this section we first show how to find a probabilistic error bound for a given level of confidence for a single set of simulation points. This can be used to estimate the confidence and error for a given clustering. We then show how to use this same analysis to create a new algorithm for picking k (the number of clusters to use). We then compare the results of this algorithm with statistical sampling.

4.1 Statistical Validation of Simulation Points

Given a set of simulation points obtained from SimPoint, a user may want to know the confidence and error for this set of simulation points. We use the following approach for estimating what the expected error is for a given level of statistical confidence for a single set of simulation points. We then show how to use this technique to guide picking k (the cluster to use) for SimPoint.

To establish the error bounds, we must find the variance of the estimator. A very simple and intuitive way to do this is to repeatedly take estimates from the program. This sampling technique is known as a parametric bootstrap [2, p. 480], where the parametric form is the clustering structure we have learned.

Recall from Section 1.1 that for the clustering k chosen by SimPoint, one simulation point (interval) is chosen to represent the entire cluster. This set of simulation points for all the clusters is then used to represent the complete execution of the program. To find the error bound for this one set of simulation points, we do the following:

1. Find a clustering using the BIC heuristic
2. Do the following N times:
   (a) Choose one interval (sample) at random from each cluster/phase.
   (b) Compute an estimated CPI by combining the CPIs from each chosen interval, weighted by the size of each cluster.
3. The probabilistic error bound on *one* set of simulation points is $z\sigma/\mu$.

Here μ is the average of the N estimated CPIs, and σ is the standard deviation of the computed estimate CPIs. The value z is the "confidence multiplier" that comes from a table of the normal distribution; $\alpha = F(z)$, where F is the cumulative distribution function of the normal distribution, so that $z = F^{-1}(\alpha)$. Thus, z is the value such that the area under the Gaussian curve to the left of z is α, the desired confidence. Then $e = z\sigma/\mu$ is the probabilistic error bound for a given level of confidence α. For a choice of *one* set of simulation points, we expect the true CPI to be within $z\sigma/\mu$ of the estimate μ.

The value of N is the number of times to compute estimates. Larger N gives a more accurate and tighter measurements of the standard deviation; $N = 10$ or larger is reasonable; for our measurements we have used $N = 100$. Note that gathering all the simulation points for the N CPI estimates only requires one run through the program. We first choose N random samples from every cluster, and then run the program once fast-forwarding between the samples to gather all of the results. Then, the above analysis can be used to determine for a given confidence level a probabilistic error bound for any SimPoint clustering.

4.2 Picking K using Variance Analysis

We now describe a new SimPoint algorithm where the user enters a desired confidence and a probabilistic error bound, and then the smallest clustering k is picked that matches these constraints. The first priority of the algorithm is to ensure that candidate clusterings are chosen ***first*** according to the homogeneity of their clusters based on code usage, and then ***second*** based upon a confidence and probabilistic error bound. This is because the confidence and error are calculated with respect to CPI and sampling a particular architecture configuration. If we did not choose a clustering based first upon code usage and instead only on confidence and error, then the clustering may not be representative across different hardware configurations.

In this algorithm, as before, we cluster the data for all possible values of K from 1 to max K that is specified by the user. To ensure that a clustering is picked that would be representative and independent of the underlying architecture we first apply the BIC heuristic to all of the clusterings. The new algorithm starts to differ here. We trim down the possible set of clusterings from K down to B. These B clusterings have a BIC score greater than a specified threshold (80% for the results in this paper). We then search this candidate set of B clusterings for the smallest k that meets the desired confidence and error. Picking a subset of clusterings based on BIC and then a final clustering based on Variance in this manner ensures that the given set of simulation points chosen will be representative of the complete execution regardless of the underlying architecture.

Our Variance SimPoint algorithm allows the user to set a desired error (e.g., 5%) within which the estimate should come to the true value, and a confidence level on that desired error (e.g., 95%). The algorithm uses sampling to determine an appropriate number of clusters for that desired error.

1. Choose a desired error e and confidence level α.
2. For each $k = \{1, \ldots, K\}$, find a clustering of the Basic Block Vectors with k clusters using the k-means algorithm. Here K is the maximum number of clusters to consider, also specified by the user.
3. Trim K down to B candidate clusterings. Candidate clusterings are those with a BIC score that is at least 80% as good as the best BIC score. This insures that we only examine clusterings that are well formed, by first only considering the similarities in the code executed between intervals. This results in a set B of *candidate clusters* and only these clusters will be considered in the rest of the algorithm.
4. From *each candidate clustering*, choose N samples randomly *from each cluster*. This results in a total of S samples to gather.
5. Run the program/input pair gathering architecture results for each of the S samples.
6. For each candidate clustering k, calculate the estimated CPI μ_k the standard deviation σ_k, and error as described above in Section 4.1. For the desired confidence level α, the probabilistic error e is calculated to be $e = z\sigma_k/\mu_k$.
7. Select the smallest k, from the set of candidate clusterings B such that $z\sigma_k/\mu_k \leq e$. This picks the smallest k that has a small enough standard deviation to satisfy the desired error bound at the given confidence level.

The above algorithm determines a set of candidate clusterings B that satisfy the above BIC heuristic. This ensures that we are only considering clusterings that are well formed. To perform the Variance analysis, we need to gather samples from the candidate clusterings. For each clustering in B, we randomly select N points from each cluster to sample. For example, assume we start with a max $K = 20$, then after applying the BIC score we are left with only three candidate clusterings, which are $k = 18, 19, 20$ (remember that $k = 18$ has 18 clusters, $k = 19$ has 19 clusters, and so on). We then have to gather $S = (18 + 19 + 20) * N$ samples to calculate the confidence and error analysis for each of these three clusterings. Once the sample locations are chosen, the program/input pair is then run once to gather all of the samples. Once this is done, we can then calculate the probabilistic error bound for the given input confidence level for each clustering k. We then chose the smallest candidate k such that $z\sigma_k/\mu_k \leq e$, using the statistical analysis described above in Section 4.1.

Now for some programs, it may be that for the max K used (limit on the number of clusters) that σ_k never reaches a small enough value to allow any clusterings to be acceptable at the desired error and confidence. In this case, several options can be followed. One option is to increase the maximum K to consider, and repeat the above process, which will cause the σ_k to decrease for the new, larger possible values of k. Another option is to use the clustering k with the lowest σ_k, understanding that the desired error will not be achieved; rather, the expected error will be within $c\sigma_k/\mu_k$ percent of the true value.

4.3 Variance SimPoint Results

Figure 1 shows the results for the Variance SimPoint algorithm using an interval (sample) size of 1 million instructions, and a max K of 300 clusters, so at most 300 million instructions would be detailed simulated when using the simulation points. The desired confidence level used for these results is for $\alpha = 0.95$, or 95% confidence with an error less than 5%. For this paper, we gathered results for 43 SPEC program/input combinations. Figure 1 shows the results for all of these for the Variance SimPoint algorithm. The percent error is shown when compared to the complete execution of the program/input, and top of the stem shows the probabilistic error bound.

Figures 2 and 3 show the number of clusters used and the number of instructions the program had to fast-forward through to get to the last simulation point chosen for the Variance SimPoint results in Figure 1. In these two figures, detailed results are shown for a subset of 10 programs, `avg-rest` shows the average for all the 33 programs not shown in detail, and then an `avg-all` shows the overall average for all 43 programs. The results show that on average 100 million instructions are simulated in detail, and there is usually a simulation point chosen somewhere near the end of the program, so the fast-forward distance is almost equal to the full execution of the program.

The second and third bars in Figure 1 show results for random sampling with (1) the same number of intervals chosen as for the Variance SimPoint algorithm, and (2) when using 1000 samples. The sample size used is also 1 million instructions. For these results, we use a form of random sampling instead of systematic sampling. To perform our random sampling, we divide the program up into N consecutive sections, where N is the number of samples we are going to take. We then randomly choose one point from each of the N sections. This guarantees that we get a random distribution of samples across the complete execution of the program/input. For the program/inputs examined, we found this to provide tighter probabilistic error bounds and lower errors than pure random sampling or systematic sampling.

The results show that Variance SimPoint is able to achieve tighter probabilistic error bounds than Sampling, when using the same number of samples. Variance SimPoint ensures that at least one sample is being used from each cluster, and the purpose of clustering is to group program behavior such that each cluster potentially represents different program behavior. Therefore, obtaining a sample of all of the different behaviors in a program allows Variance SimPoint to achieve tight error bounds when using a small number of samples (on average 100 as shown in Figure 2). In comparison, when using the same number of samples for each program, random sampling has a worse probabilistic error bound. This is because random sampling can oversample some of the program's behavior while under-sampling other parts of the program. When using 1000 samples, less than a 2%

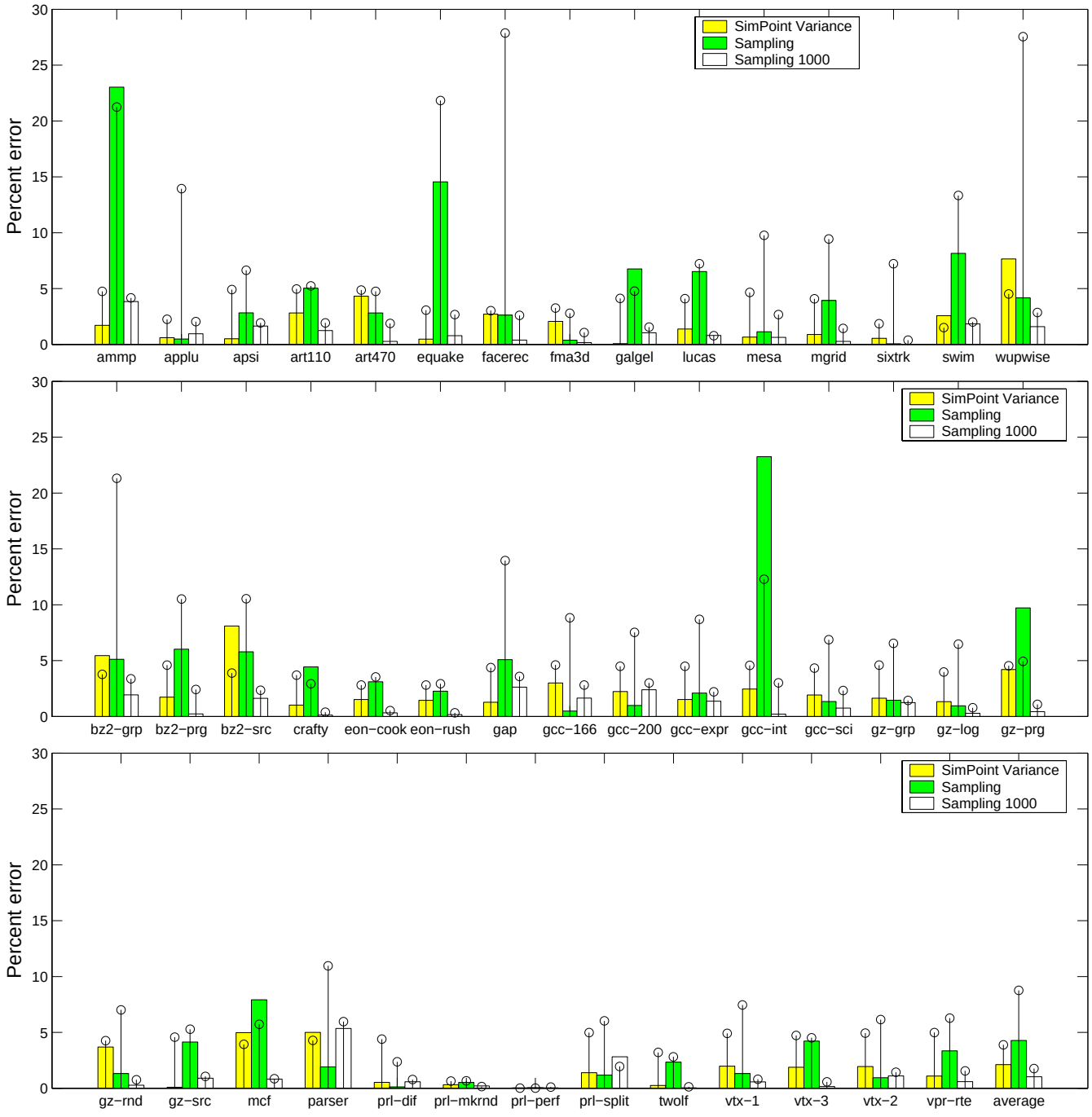


Figure 1: True CPI error (bars) and probabilistic error bounds (stems) for SimPoint using variance to choose k (the clustering). Variance SimPoint is compared to random sampling using the same number of samples as SimPoint, and random sampling with 1000 samples. The probabilistic error bounds are for 95% confidence.

average error is seen with under a 3% probabilistic error bound.

5 Early Simulation Points

In the original multiple SimPoint algorithm [19], the goal was to pick a single simulation point from each cluster that best represents all of the intervals in that cluster. This may pick simulation points that are at the very end of a program's execution. If the simulator supports checkpointing, then simulation can be started very quickly at a point at the end of the program. But, for simulation environments that do not support checkpointing, it can require up to several days to fast-forward to the latter part of execution to reach a late simulation point.

The goal in this section is to find simulation points that are earlier in the program's execution that still accurately represent the overall execution of the program. These early simulation points can then be used to significantly reduce the time spent fast-forwarding to reach all of the simulation points for a program.

5.1 Early SimPoint Algorithm

This section focuses on a simulation environment that relies upon using fast-forwarding to simulate a program. In this environment, the program is simulated once interleaving fast-forwarding with detailed simulation. The *last* simulation point in a program's execution determines how much of the program the simulator will have to fast-forward through, and this greatly determines the total simulation time. Therefore, to reduce the time required for fast-forwarding, we only care about the location of the last simulation point. If we pick the earliest point from each cluster, then the earliest the last simulation point in the program will occur is the location of the latest starting cluster.

Our Early SimPoint algorithm focuses on choosing a clustering that is both representative of the program's execution and has some feasible simulation points early in the program for all clusters. This might not be achievable for all programs, since an important phase of execution may only appear at the end of execution. We therefore still give priority in our algorithm to ensure that the clustering groups together intervals of execution that are similar to one another. Once the early clustering is chosen, we pick representative simulation points early in the execution from all the clusters.

5.1.1 Picking a Clustering

As described in Section 1.1, the original SimPoint algorithm uses k-means to perform several clusterings for different values of k. It then uses the BIC score to choose a clustering, and then from each cluster the centroid is picked to represent that cluster. For the Early SimPoint algorithm, we perform the exact same k-means clustering algorithm as in the original SimPoint algorithm. The Early algorithm differs in how it chooses which clustering to use, and then how it chooses which points to use from each cluster.

In picking a cluster (k), the Early SimPoint algorithm takes into consideration where the intervals for a given cluster are located over time (the execution of the program). The goal is to pick a clustering, where all clusters have some intervals early in the program's execution, while still clustering together similar intervals of execution.

To guide this we introduce a new metric, *EarlySP*, which is the BIC score weighted by the first encounter of the last cluster: $EarlySP = BIC \times (1 - (StartLastCluster/w))$. The intuition behind *EarlySP* is that we reward the clusterings that have representatives from every cluster near the start of the program. $StartLastCluster$ is the percent into execution of the program that the last cluster is first encountered. For example, if for a clustering we have a $StartLastCluster$ of 40%, this means that one of the clusters has its first interval of execution occurring 40% into execution, and all of the rest of the clusters have at least one interval earlier in the execution than this. The $StartLastCluster$ is critical for picking early simulation points, since it will be the minimum distance required to fast-forward.

The variable w is a weight to influence the impact of how early the last cluster is on the BIC. Since we prioritize accurate program representation over early simulation, the variable w limits how much the BIC can be influenced by the $StartLastCluster$ term. Setting w to 10 guarantees that we remain within 10% of the true BIC, ensuring representative scoring is maintained for the clusterings.

The *EarlySP* score provides us with a goodness of fit of the clustering weighted by how early the last cluster starts. We score each clustering generated and then pick the smallest k that achieves at least 80% of the spread between the largest and smallest *EarlySP* scores. We can set the threshold higher if we want tighter clusters, at the cost of having more simulation points. This tradeoff is illustrated in the following Section 5.2.

5.1.2 Picking Simulation Points

After Early SimPoint picks a clustering, we determine a cutoff point in the program's execution and we consider picking simulation points only from the start of the program to this cutoff point. No intervals of execution after this cutoff point will be considered for simulation points. This bounds the amount of overhead due to fast-forwarding.

The cutoff point for a program/input is determined by first picking an early simulation point for the cluster that starts the latest in execution. The cutoff simulation point is then picked for the last cluster. To pick this cutoff point, we first sorted all of the simulation points for the latest starting cluster using each point's Euclidean distance from the centroid. Then the 1% closest points are candidates for being the cutoff simulation point. The earliest of these in the program's execution is then chosen to be the cutoff simulation point. Once this cutoff simulation point has been determined, the simulation points for the remaining clusters are selected from all the potential intervals from the start of the program up to the cutoff interval. For a given cluster, a simulation point is chosen from these candidate intervals that is closest to the original centroid of the cluster.

5.2 Early SimPoint Results

To examine the performance of Early SimPoint we compare the results from five different algorithms. To evaluate these different algorithms, Figure 2 shows the number of clusters (the value

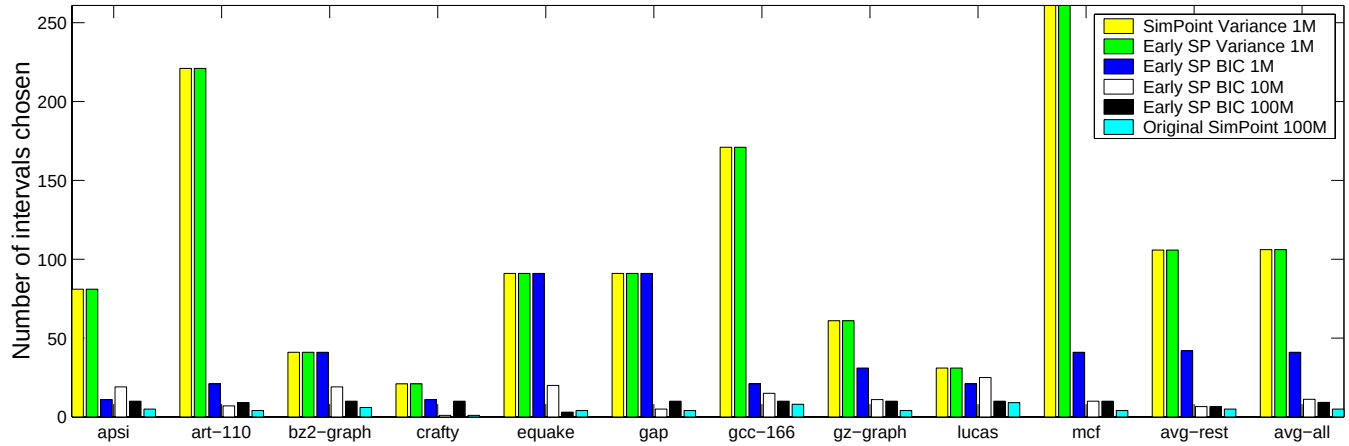


Figure 2: Number of simulation points used for each program/input for the different SimPoint algorithms. The number of simulation points is equivalent to the value of k chosen, which is the number of clusters. A single simulation point is picked from each cluster.

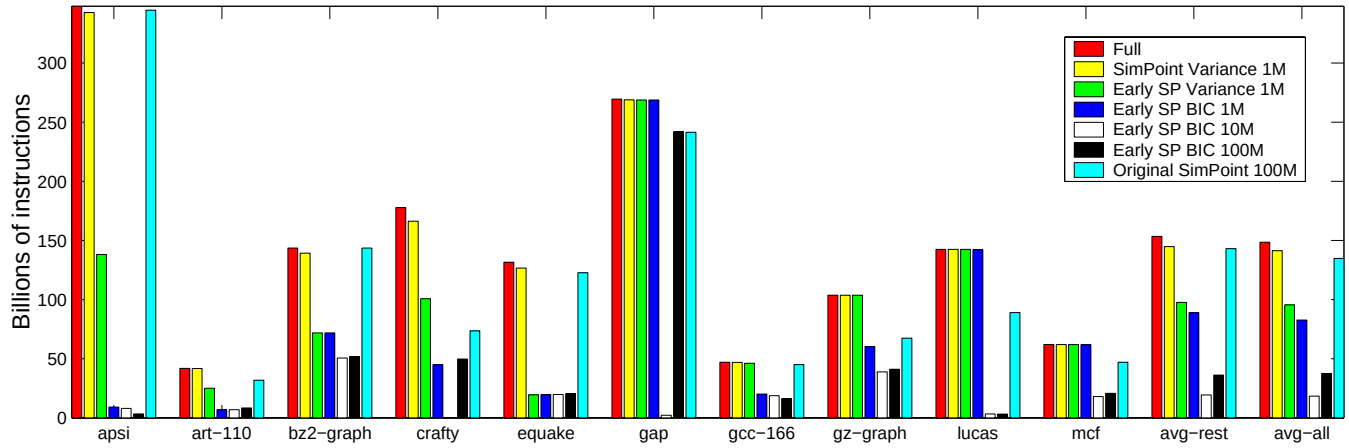


Figure 3: The number of instructions in Billions needed for fast-forwarding to reach the last simulation point for the different SimPoint algorithms. The results for Full show the length of the complete execution of each program/input run. The goal of Early SimPoint is to reduce the amount of fast-forwarding by picking representative, but early, simulation points.

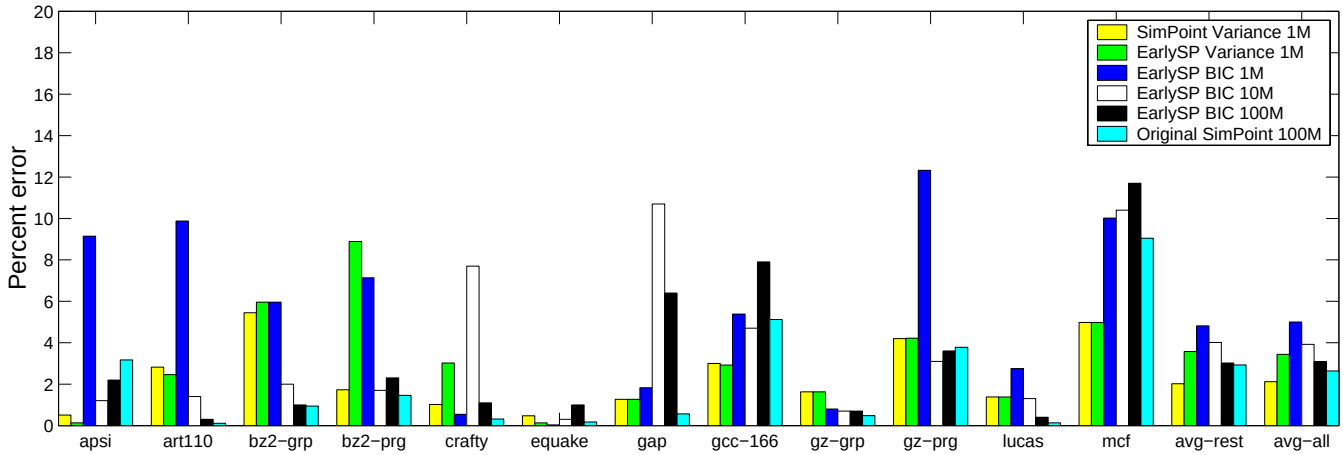


Figure 4: Relative CPI error to the full simulation for the different SimPoint algorithms varying the interval size over 1, 10, and 100 Million sample sizes.

chosen for k) that was picked for each algorithm, and this equals the number of simulation points. Figure 3 shows the number of instructions (in billions) it will take from the beginning of execution to reach the last simulation point. This number, minus the number of detailed instructions simulated, is the number of instructions fast-forwarded to simulate each program. Finally, Figure 4 shows the error rates for the different algorithms.

Different algorithms have results shown using an interval size of 1 million (1M), 10 million (10M), and 100 million (100M) instructions. Recall from Section 1.1, that this interval size is the granularity in which the basic block vector (code) profile is collected and the clustering is performed. It is also the length of the simulation point when performing detailed simulation. Due to space, we only show a variety of different combinations of algorithm and interval size.

The 100 million interval size results show the performance when using the original BIC SimPoint algorithm from [19] and the Early SimPoint algorithm described in this section. For both results max K is set to be 10 intervals to put a limit on simulation time. We provide results for the original SimPoint algorithm using a BIC threshold of 80%. For Early SimPoint results, we use an EarlySP threshold of 100%. Setting EarlySP to have a threshold of 100% picks the clustering that achieves the highest EarlySP score. We use this for the 100M interval results, since only a small number of clusterings (max $K = 10$) are examined. With having at max 10 clusters, complex programs will be noisier within a cluster, so picking the best scoring clustering ensures that the clusters are as well formed as possible with early simulation points. The results show that the number of instructions required for fast-forwarding for Early SimPoint is 3.6 times smaller than using the original SimPoint algorithm. This comes at the cost of increasing the average error from 2.6% to 3.1%.

We also provide results for Early SimPoint using an interval size of 10 million and 1 million. For both of these results, a EarlySP threshold of 80% was used. The results for 1 million intervals shows that it has a fast-forward length 5 times longer than the 10 million interval size. This is because there are more clusters, which creates a greater chance of a cluster showing up only late in the execution. The 10 million interval size results show that they require the least amount of fast-forward time, with the fast-forward length being only 13% into the program/input run's execution on average. These results show a delicate trade-off between speed and accuracy, as well as choosing an appropriate interval size.

All of the results we have talked about to this point (Original SimPoint and Early SimPoint algorithms) in this section are based only on the BIC, and do not use any confidence or probabilistic error bounds to guide the choosing of a clustering. We now look at using the Early SimPoint approach with the Variance SimPoint algorithm. When using EarlySP with the Variance algorithm in Section 4, we use the Variance algorithm for picking k just as before, and then use the approach described in Section 5.1.2 to pick the simulation points. The Variance SimPoint is performed, and then simulation points are only chosen from intervals that occur from the start of execution to the simulation point of the latest starting cluster.

Figure 4 shows that picking early simulation points in the Variance SimPoint algorithm achieves an error rate of 3.4%, which is close to the non-early Variance SimPoint algorithm error of 2.1%. We also found that it had a very similar error bound (not shown on the figure) of 3.9% on average for 95% confidence. Figure 3 shows that the non-early Variance SimPoint algorithm has a fast-forward length 1.5 times longer than when picking early points for the Variance SimPoint algorithm. With 100 simulation points chosen on average for these results, the likelihood of clusters appearing only in the latter portion of execution is significant. Even with 40 samples, as is the case for the early BIC approach at 1M interval size, the ability to find all the clusters early in the execution is low.

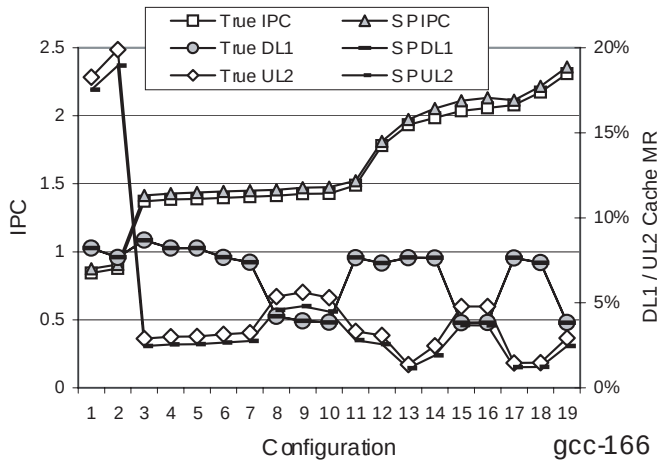
6 Tracking Metrics Across Architecture Configurations

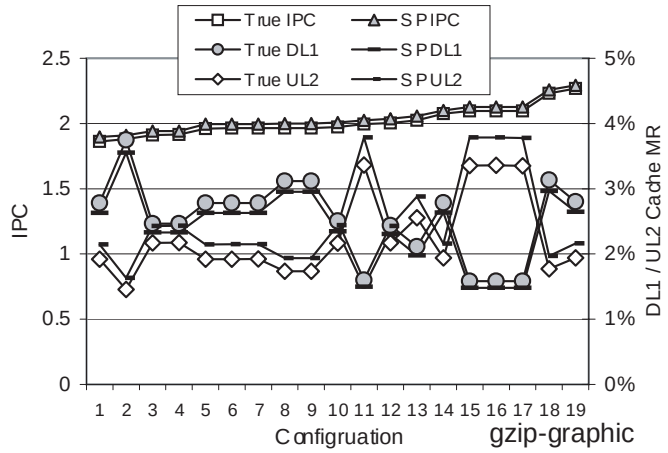
The absolute error of a program/input run on one hardware configuration is not as important as tracking the change in metrics across different architecture configurations. In this section we show that SimPoint accurately tracks the relative change in hardware metrics across several different architecture configurations. We show that using the same set of simulation points from SimPoint results in a bias error that is consistent (not random) across the different configurations.

We show that our simulation points, which are independent from the underlying architecture, are representative of the full program's execution when varying the architecture parameters. The key insight for why this works is that the BIC SimPoint algorithm for clustering, choosing a cluster, and picking the simulation points is based only on the code usage over the program's execution not on any architecture metrics.

To examine the independence of our simulation points from the underlying architecture, we used the simulation points for the BIC SimPoint algorithm with 1 million intervals and max K set to 300. For the program/input runs we examine, we performed full program simulations varying the memory hierarchy, and for every run we used the same set of simulation points when calculated the SimPoint estimates. We varied the configurations and the latencies of the L1 and L2 caches. As we increased the size and the associativity of the caches we increased their latency to model the effect of architecture scaling. We chose L1 instruction and data cache sizes between 4 KBytes and 64 KBytes, varied the associativity between direct mapped and 4-way associative, and varied their latency from 1 to 3 cycles. At the same time we varied the size of the unified L2 cache from 500 KBytes to 2 MBytes, its associativity from 4-way to 8-way associative, and varied the latency from 10 to 40 cycles.

Figure 5 shows the results across the 19 different architecture configurations we examined for three programs from the SPEC benchmark suite `gcc-166`, `gzip-graphic` and `art-110`. The left y-axis represents the performance in $InstructionsPerCycle$ and the x-axis represents different memory configurations from the baseline architecture. The right y-axis shows the miss rates for the data cache and unified L2





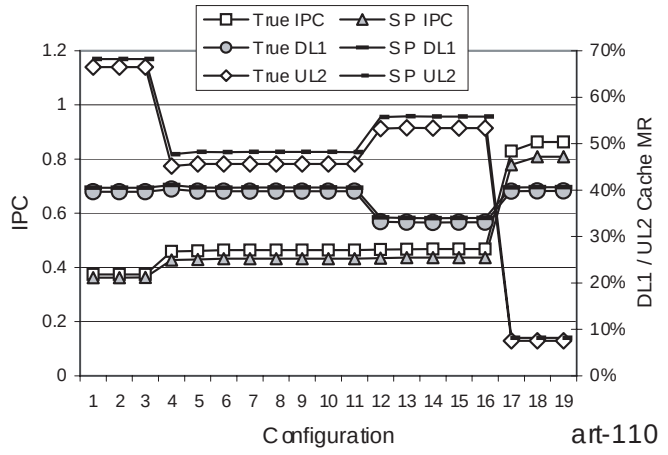


Figure 5: This plot shows on the left y-axis the true and estimated IPC for `gcc`, `gzip`, *and* `art` *for 19 different architecture configurations. The right y-axis shows each configurations corresponding miss rates for the L1 data cache and unified L2 cache. Results are shown for the complete execution of the configuration, and when using SimPoint.*

cache, and the L2 miss rate is a local miss rate. For each metric, two lines are shown, one for the true metric from the *complete* detailed simulation for every configuration, and the second for the estimated metric using our simulation points. For each graph, the configurations on the x-axis are sorted by the IPC of the full run.

Figure 5 shows that the simulation points from using SimPoint creates a bias in the metrics, and this bias is consistent and always in the same direction across the different configurations for a given program/input run. This is true for IPC and the cache miss rates. This shows that simulation points, which are chosen by only looking at code usage, can be used across different architecture configurations to make accurate architecture design trade-off decisions and comparisons.

The absolute error of one configuration is not as important as tracking the change in a metric across different architecture configurations. Our results show that simulation points track the relative change in metrics between configurations. When there is a change between two configurations, that change is clearly seen using SimPoint in Figure 5, and even small changes are accurately tracked.

One way to evaluate this is to rank the configurations using the metrics from the full execution and then rank the configurations using the SimPoint estimates of the metrics and see if the rankings are the same. We separately performed the configuration ranking for all of the hardware metrics we examined IPC, L1 and L2 cache miss rates, and in every case the ranking of the hardware configurations were in the exact same order when using the full metric and the SimPoint estimates. This is easily seen in Figure 5, where the configurations in the x-axis is sorted by the IPC of the full simulation. Overall, these results show that SimPoint can be used to accurately prune the choices when searching the architecture design space.

7 Summary

In [18, 19], we presented the idea of profiling the code usage over the execution of a program and using this to cluster intervals of execution together that had similar behavior. A single point was then picked to represent each cluster, and these simulation points were used to guide accurate and efficient simulation. The research in this paper addresses three open issues from our original algorithm. These are to (1) provide statistical validation with confidence and error for a given clustering, and to use this analysis to help choose which clustering to use, (2) create a method for generating simulation points earlier in execution, and (3) showing that the points generated from SimPoint accurately track metrics between different architecture configurations.

In this paper, we first present the Variance SimPoint algorithm that uses a user defined confidence and probabilistic error bound to guide the picking of k. This algorithm first gives priority to choosing a clustering that has well formed (based upon the code frequencies) clusters. This is to make sure that the clustering is representative across different architecture configurations. The Variance SimPoint clustering for picking samples shows that tighter probabilistic error bounds are seen when com-

pared to random sampling when using the same amount of samples as there are simulation points. This is due to Variance SimPoint always choosing a representative sample from each cluster to make sure all the unique/different behavior in the program is captured. In comparison, for a small number of samples, random sampling tends to over/under sample some of the program's behavior resulting in looser probabilistic error bounds.

The second contribution of the paper is the Early SimPoint algorithm whose goal is to find representative simulation points early in a program's execution to reduce fast-forward simulation time. The amount of fast-forward time required for a simulation is the number of instructions it takes to reach the last simulation point for a program/input run. When using an execution interval size of 100 million instructions, we found that Early SimPoint had a 3.6 times shorter fast-forwarded length on average than the original SimPoint algorithm. When using an interval size of 10 million, we found the fast-forward length to be only 12% of the full execution on average, with an average CPI error of 4%. These results show that these early simulation points can be used to significantly reduce the time spent fast-forwarding to reach all of the simulation points for a program, while still providing accurate results.

The final contribution was to examine the ability of simulation points to track metrics across architecture changes. The absolute error of a program/input run on one hardware configuration is not as important as tracking the change in metrics across different architecture configurations. We showed that SimPoint accurately tracks the relative change in hardware metrics (CPI and cache miss rates) across several different architecture configurations. Using the same set of simulation points results in a bias error that is consistent (not random) across the different configurations. These results show that the SimPoint simulation points, which are chosen by only looking at code usage, can be used across different architecture configurations to make accurate architecture design trade-off decisions and comparisons.

Acknowledgments

We would like to thank Tim Sherwood and Tom Conte for their insights and help, and the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, Semiconductor Research Corporation grant No. SRC-2001-HJ-897, and an equipment grant from Intel.

References

[1] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury, Pacific Grove, CA, USA, 2002.

[3] T. M. Conte, M. A. Hirsch, and W. W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6):714–720, 1998.

[4] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[5] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[6] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, September 2001.

[7] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerating sampled microarchitecture simulations. Technical Report CS-2002-19, U of Virginia, July 2002.

[8] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.

[9] R. E. Kass and L. Wasserman. A reference Bayesian test for nested hypotheses and its relationship to the schwarz criterion. *Journal of the American Statistical Association*, 90(431):928–934, 1995.

[10] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[11] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the International Conference on Computer Design*, September 2000.

[12] S. Laha, J. A. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11), February 1988.

[13] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical Report SMLI TR-93-22, Sun Microsystems Laboratories, December 1993.

[14] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[15] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[16] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[17] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[18] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming*, October 2002. `http://www.cs.ucsd.edu/users/calder/simpoint/`

[20] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[21] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. *1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems ACM SIGMETRICS Performance Evaluation Review*, 19(1), May 21-24, 1991.

[22] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture*, June 2003.