Journal of Instruction-Level Parallelism 7 (2005) 1-28 Submitted 6/2005; published 9/2005

SimPoint 3.0: Faster and More Flexible Program Phase Analysis

Greg Hamerly HAMERLY@CS.ECS.BAYLOR.EDU
Baylor University
One Bear Place #97356
Waco, TX 76798-7356, USA

Erez Perelman EPERELMA@CS.UCSD.EDU
Jeremy Lau JL@CS.UCSD.EDU
Brad Calder CALDER@CS.UCSD.EDU
University of California, San Diego
9500 Gilman Drive
La Jolla CA 92093-0404 USA

Abstract

This paper describes the new features available in the SimPoint 3.0 release. The release provides two techniques for drastically reducing the run-time of SimPoint: faster searching to find the best clustering, and efficiently clustering large numbers of intervals. SimPoint 3.0 also provides an option to output only the simulation points that represent the majority of execution, which can reduce simulation time without much increase in error. Finally, this release provides support for correctly clustering variable length intervals, taking into consideration the weight of each interval during clustering. This paper describes SimPoint 3.0's new features, how to use them, and points out some common pitfalls.

1. Introduction

Modern computer architecture research requires understanding the cycle level behavior of a processor during the execution of an application. To gain this understanding, researchers typically employ detailed simulators that model each and every cycle. Unfortunately, this level of detail comes at the cost of speed, and simulating the full execution of an industry standard benchmark can take weeks or months to complete, even on the fastest of simulators. To make matters worse, architecture researchers often simulate each benchmark over a variety of architecture configurations and designs to find the set of features that provide the best trade-off between performance, complexity, area, and power. For example, the same program binary, with the exact same input, may be run hundreds or thousands of times to examine how the effectiveness of an architecture changes with cache size. Researchers need techniques to reduce the number of machine-months required to estimate the impact of an architectural modification without introducing an unacceptable amount of error or excessive simulator complexity.

At run-time, programs exhibit repetitive behaviors that change over time. These behavior patterns provide an opportunity to reduce simulation time. By identifying each of the repetitive behaviors and then taking only a single sample of each repeating behavior,

we can perform very fast and accurate sampling. All of these representative samples together represent the complete execution of the program. The underlying philosophy of SimPoint [1, 2, 3, 4, 5, 6] is to use a program's behavior patterns to guide sample selection. SimPoint intelligently chooses a very small set of samples called *Simulation Points* that, when simulated and weighed appropriately, provide an accurate picture of the complete execution of the program. Simulating only these carefully chosen simulation points can save hours to days of simulation time with very low error rates. The goal is to run SimPoint once for a binary/input combination, and then use these simulation points over and over again (potentially for thousands of simulations) when performing a design space exploration.

This paper describes the new SimPoint 3.0 release. In Section 2 we present an overview of the SimPoint approach. Section 4 describes the new SimPoint features, and describes how and when to tune these parameters. It also provides a summary of SimPoint's results and discusses some suggested configurations. Section 5 discusses the common pitfalls to watch for when using SimPoint, and Section 6 summarizes this paper. Finally, the appendix describes in detail the command line options for SimPoint 3.0.

The major new features for the SimPoint 3.0 release include:

- **Efficient searching to find the best clustering.** Instead of trying every value, or every Nth value, of k when running the k-means algorithm, we provide a binary search method for choosing k. This typically reduces the execution time of SimPoint by a factor of 10.

- **Faster SimPoint analysis when processing many intervals.** To speed the execution of SimPoint on very large inputs (100s of thousands to millions of intervals), we sub-sample the set of intervals that will be clustered. After clustering, the intervals not selected for clustering are assigned to phases based on their nearest cluster.

- **Support for Variable Length Intervals.** Prior versions of SimPoint assumed fixed length intervals, where each interval represents the same amount of dynamic execution. For example, in the past, each interval represented 1, 10, or 100 million dynamic instructions. SimPoint 3.0 provides support for clustering variable length intervals, where each interval can represent different amounts of dynamic execution. With variable length intervals, the weight of each interval must be considered during clustering.

- **Reduce the number of simulation points by representing only the majority of executed instructions.** We provide an option to output only the simulation points whose clusters account for the majority of execution. This reduces simulation time, without much increase in error.

2. Background

Several other researchers have worked on phase analysis, and we review some of the related work here.

2.1 Related Work on Phase Analysis

The recurring use of different areas of memory in a program is first noted by Denning and Schwarz [7] and is formalized as the idea of working sets. While working sets have driven the development of caches for decades, recently many of the more subtle implications of recurring behaviors have been explored by researchers in the computer architecture community.

Balasubramonian *et al.* [8] proposed using hardware counters to collect miss rates, CPI and branch frequency information for every 100,000 instructions. They use these miss rates and the total number of branches executed for each interval to dynamically evaluate the program's stability. They used their approach to guide dynamic cache reconfiguration to save energy without sacrificing performance.

Dhodapkar and Smith [9, 10, 11] found a relationship between phases and instruction working sets, and show that phase changes occur when the working set changes. They proposed a method by which the dynamic reconfiguration of multi-configuration units can be controlled in response to phase changes indicated by working set changes. Through a working set analysis of the instruction cache, data cache and branch predictor they derive methods to save energy.

Hind *et al.* [12] provide a framework for defining and reasoning about program phase classifications, focusing on how to best define granularity and similarity to perform phase analysis.

Isci and Martonosi [13, 14] have shown the ability to dynamically identify the power phase behavior using power vectors. Deusterwald *et al.* [15] recently used hardware counters and other phase prediction architectures to find phase behavior.

These related methods offer alternative techniques for representing programs for the purpose of finding phase behaviors. They each also offer methods for using the data to find phases. Our work on SimPoint frames the problem as a clustering problem in the machine learning setting, using data clustering algorithms to find related program behaviors. This problem is a natural application of data clustering, and works well.

2.2 Phase Vocabulary

To ground our discussion in a common vocabulary, the following is a list of definitions we use to describe the analysis performed by SimPoint.

- Interval - A section of continuous execution (a slice in time) of a program. All intervals are assumed to be non-overlapping, so to perform our analysis we break a program's execution into contiguous non-overlapping intervals. The prior versions of SimPoint required all intervals to be the same size, as measured in the number of instructions committed within an interval (e.g., interval sizes of 1, 10, or 100 million instructions were used in [3]). SimPoint 3.0 still supports fixed length intervals, but also provides support for *Variable Length Intervals* (VLI), which allows the intervals to account for different amounts of executed instructions as described in [16].

- Phase - A set of intervals within a program's execution with similar behavior. A phase can consist of intervals that are not temporally contiguous, so a phase can re-appear many times throughout execution.

- Similarity - Similarity defines how close the behavior of two intervals are to one another as measured across some set of metrics. Well-formed phases should have intervals with similar behavior across various architecture metrics (e.g. IPC, cache misses, branch misprediction).

- Frequency Vector - Each interval is represented by a frequency vector, which represents the program's execution during that interval. The most commonly used frequency vector is the basic block vector [1], which represents how many times each basic block is executed in an interval. Frequency vectors can also be used to track other code structures [5] such as all branch edges, loops, procedures, registers, opcodes, data, or program working set behavior [17] as long as tracking usage of the structure provides a signature of the program's behavior.

- Similarity Metric - Similarity between two intervals is calculated by taking the distance between the corresponding frequency vectors from the two intervals. SimPoint determines similarity by calculating the Euclidean distance between the two vectors.

- Phase Classification - Phase classification groups intervals into phases with similar behavior, based on a similarity metric. Phase classifications are specific to a program binary running a particular input (a binary/input pair).

2.3 Similarity Metric - Distance Between Code Signatures

SimPoint represents intervals with frequency vectors. A frequency vector is a one dimensional array, where each element in the array tracks usage of some way to represent the program's behavior. We focus on code structures, but a frequency vector can consist of any structure (e.g., data working sets, data stride access patterns [5, 17]) that may provide a signature of the program's behavior. A frequency vector is collected from each interval. At the beginning of each interval we start with a frequency vector containing all zeros, and as the program executes, we update the current frequency vector as structures are used.

A common frequency vector we have used is a list of static basic blocks [1] (called a Basic Block Vector (BBV)). If we are tracking basic block usage with frequency vectors, we count the number of times each basic block in the program has been entered in the current interval, and we record that count in the frequency vector, weighted by the number of instructions in the basic block. Each element in the frequency vector is a count of how many times the corresponding basic block has been entered in the corresponding interval of execution, multiplied by the number of instructions in that basic block.

We use basic block vectors (BBV) for the results in this paper. The intuition behind this is that the behavior of the program at a given time is directly related to the code executed during that interval [1]. We use the basic block vectors as signatures for each interval of execution: each vector tells us what portions of code are executed, and how frequently those portions of code are executed. By comparing the BBVs of two intervals, we can evaluate the similarity of the two intervals. If two intervals have similar BBVs, then the two intervals spend about the same amount of time in roughly the same code, and therefore we expect the behavior of those two intervals to be similar. Prior work showed that loop and procedure vectors can also be used, where each entry represents the number

of times a loop or procedure was executed, performs comparably to basic block vectors [5], while using fewer dimensions.

To compare two frequency vectors, SimPoint 3.0 uses the Euclidean distance, which has been shown to be effective for off-line phase analysis [2, 3]. The Euclidean distance is calculated by viewing each vector as a point in D-dimensional space, and calculating the straight-line distance between the two points.

2.4 Using k-Means for Phase Classification

Clustering divides a set of points into groups, or clusters, such that points within each cluster are similar to one another (by some metric, usually distance), and points in different clusters are different from one another. The k-means algorithm [18] is an efficient and well-known clustering algorithm which we use to quickly and accurately split program behavior into phases. The k in k-means refers to the number of clusters (phases) the algorithm will search for.

The following steps summarize the phase clustering algorithm at a high level. We refer the interested reader to [2] for a more detailed description of each step.

1. Profile the program by dividing the program's execution into contiguous intervals, and record a frequency vector for each interval. Each frequency vector is normalized so that the sum of all the elements equals 1.

2. Reduce the dimensionality of the frequency vector data to a smaller number of dimensions using random linear projection.

3. Run the k-means clustering algorithm on the reduced-dimension data for a set of k values.

4. Choose from among these different clusterings a well-formed clustering that also has a small number of clusters. To compare and evaluate the different clusters formed for different values of k, we use the *Bayesian Information Criterion* (BIC) [19] as a measure of the "goodness of fit" of a clustering to a dataset. We choose the clustering with the smallest k, such that its BIC score is close to the best score that has been seen. Here "close" means it is above some percentage of the range of scores that have been seen. The chosen clustering represents our final grouping of intervals into phases.

5. The final step is to select the simulation points for the chosen clustering. For each cluster (phase), we choose one representative interval that will be simulated in detail to represent the behavior of the whole cluster. By simulating *only* one representative interval per phase we can extrapolate and capture the behavior of the entire program. To choose a representative, SimPoint picks the interval in each cluster that is closest to the *centroid* (center) of each cluster. Each simulation point also has an associated weight, which reflects the fraction of executed instructions that cluster represents.

6. With the weights and the detailed simulation results of each simulation point, we compute a weighted average for the architecture metric of interest (CPI, miss rate, etc.). This weighted average of the simulation points gives an accurate representation of the complete execution of the program/input pair.

I Cache	16k 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
Main Memory	150 cycle latency
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 Disambig	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.

3. Methodology

We performed our analysis for the complete set of SPEC2000 programs for multiple inputs using the Alpha binaries on the SimpleScalar website. We collect all of the frequency vector profiles (basic block vectors) using SimpleScalar [20]. To generate our baseline fixed length interval results, all programs were executed from start to completion using SimpleScalar. The baseline microarchitecture model is detailed in Table 1.

To examine the accuracy of our approach we provide results in terms of CPI error and k-means variance. CPI error is the percent error in CPI between using simulation points from SimPoint and the baseline CPI of the complete execution of the program.

The k-means variance is the average squared distance between every vector and its closest center. Lower variances are better. When sub-sampling, we still report the variance based on every vector (not just the sub-sampled ones). The relative k-means variance reported in the experiments is measured on a per-input basis as the ratio of the k-means variance observed for clustering on a sample to the k-means variance observed for clustering on the whole input.

4. SimPoint 3.0 Features

In this section we describe and analyze the SimPoint features that affect the running time of the SimPoint algorithm, and the resulting simulation time and accuracy of the simulation points.

4.1 Choosing an Interval Size

When using SimPoint one of the first decisions to make is the interval size. The interval size along with the number of simulation points chosen by SimPoint will determine the simulation time of a binary/input combination. Larger intervals allow more aggregation of profile information, allowing SimPoint to search for large scale repeating behavior. In

comparison, smaller intervals allow for more fine-grained representations and searching for smaller scale repeating behavior.

The interval size affects the number of simulation points; with smaller intervals more simulation points are needed than when using larger intervals to represent the same proportion of a program. We showed that using smaller interval sizes (1 million or 10 million) results in high accuracy with reasonable simulation limits [3]. The disadvantage is that with smaller interval sizes warmup becomes more of an issue, but there are efficient techniques to address warmup as discussed in [21, 22]. In comparison, warmup is not really an issue with larger interval sizes, and this may be preferred for some simulation environments [23]. For all of the results in this paper we use an interval size of 10 million instructions.

4.1.1 Support for Variable Length Intervals

Ideally we should align interval boundaries with the code structure of a program. In [24], we examine an algorithm to produce variable length intervals aligned with the procedure call, return and loop transition boundaries found in code. A *Variable Length Interval* (VLI) is represented by a frequency vector as before, but each interval's frequency vector can account for different amounts of the program's execution.

To be able to pick simulation points with these VLIs, we need to change the way we do our SimPoint clustering to include the different weights for these intervals. SimPoint 3.0 supports VLIs, and all of the detailed changes are described in [16]. At a high level the changes focused around the following three parts of the SimPoint algorithm:

- Computing k-means cluster centers – With variable length intervals, we want the k-means cluster centers to represent the centroid of the intervals in the cluster, based on the weights of each interval. Thus k-means must include the interval weights when calculating the cluster's center. This is an important modification to allow k-means to better model those intervals that represent a larger proportion of the program.

- Choosing the Best Clustering with the BIC – The BIC criterion is the log-likelihood of the clustering of the data, minus a complexity penalty. The likelihood calculation sums a contribution from each interval, so larger intervals should have greater influence, and we modify the calculation to include the weights of the intervals. This modification does not change the BIC calculated for fixed-length intervals.

- Computing cluster centers for choosing the simulation points – Similar to the above, the centroids should be weighted by how much execution each interval in the cluster accounts for.

When using VLIs, the format of the frequency vector files is the same as before. A user can either allow SimPoint to determine the weight of each interval or specify the weights themselves (see the options in the Appendix). If the user allows SimPoint to determine the weights automatically, SimPoint will assign the weights from the frequency vector counts. For example, if one frequency vector summed to 100, and another summed to 200, then the second would have twice as much weight as the first. However, the default behavior of SimPoint is to assume fixed-length vectors, and give all vectors equal weight.

4.2 Methods for Reducing the Run-Time of K-Means

Even though SimPoint only needs to be run once per binary/input combination, we still want a fast clustering algorithm that produces accurate simulation points. To address the run-time of SimPoint, we first look at three options that can greatly affect the running time of a single run of k-means. The three options are the number of intervals to cluster, the size (dimension) of the intervals being clustered, and the number of iterations it takes to perform a clustering.

To start we first examine how the number of intervals affects the running time of the SimPoint algorithm. Figure 1 shows the time in seconds for running SimPoint varying the number of intervals (vectors) as we vary the number of clusters (value of k). For this experiment, the interval vectors are randomly generated from uniformly random noise in 15 dimensions.

The results show that as the number of vectors and clusters increases, so does the amount of time required to cluster the data. The first graphs show that for 100,000 vectors and $k = 128$, it took about 3.5 minutes for SimPoint 3.0 to perform the clustering. It is clear that the number of vectors clustered and the value of k both have a large effect on the run-time of SimPoint. The run-time changes linearly with the number of clusters and the number of vectors. Also, we can see that dividing the time by the multiplication of the number of iterations, clusters, and vectors to provide the *time per basic operation* continues to give improving performance for larger k.

4.2.1 Number of Intervals and Sub-sampling

The k-means algorithm is fast: each iteration has run-time that is linear in the number of clusters, and the dimensionality. However, since k-means is an iterative algorithm, many iterations may be required to reach convergence. We already found in prior work [2], and revisit in Section 4.2.2 that we can reduce the number of dimensions down to 15 and still maintain the SimPoint's clustering accuracy. Therefore, the main influence on execution time for SimPoint is the number of intervals.

To show this effect, Table 2 shows the SimPoint running time for `gcc-166` and `crafty-ref`, which are at the lower and upper ranges for the number of intervals and basic block vectors seen in SPEC 2000 with an interval size of 10 million instructions. The second and third column shows the number of intervals (vectors) and the original number of dimensions for each vector (these are projected down to 15 dimensions when performing the clustering). The last three columns show the time it took to execute SimPoint searching for the best clustering from k=1 to 100, with 5 random initializations (seeds) per k. SP2 is the time it took for SimPoint 2.0. The second to last column shows the time it took to run SimPoint 3.0 when searching over all k in the same manner as SimPoint 2.0, and the last column shows the clustering time when using our new binary search described in Section 4.4.3. The results show that increasing the number of intervals by 4 times increased the running time of SimPoint around 10 times. The results show that we significantly reduced the running time for SimPoint 3.0, and that combined with the new binary search functionality results in 10x to 50x faster choosing of simulation points over SimPoint 2.0. The results also show that the number of intervals clustered has a large impact on the running time of SimPoint, since it can take many iterations to converge, which is the case for `crafty`.

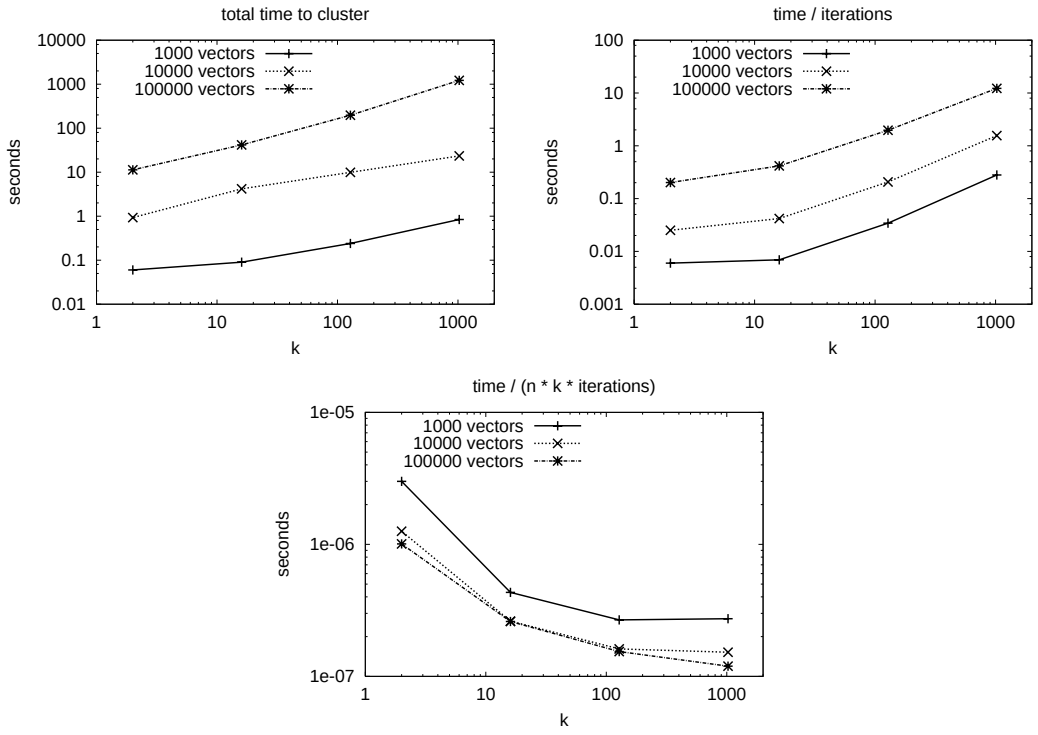

Figure 1: These plots show how varying the number of vectors and clusters affects the amount of time required to cluster with SimPoint 3.0. For this experiment we generated uniformly random data in 15 dimensions. The first plot shows total time, the second plot shows the time normalized by the number of iterations performed, and the third plot shows the time normalized by the number of operations performed. Both the number of vectors and the number of clusters have a linear influence on the run-time of k-means.

The effect of the number of intervals on the running time of SimPoint becomes critical when using very small interval sizes like 1 million instructions or smaller, where there can be millions of intervals to cluster. To speed the execution of SimPoint on these very large inputs, we sub-sample the set of intervals that will be clustered, and run k-means on only this sample. We sample the vector dataset using weighted sampling for VLIs, and uniform sampling for fixed-length vectors. The number of desired intervals is specified, and then SimPoint chooses that many intervals (without replacement). The probability of each interval being chosen is proportional to the weight of its interval (the number of dynamically executed instructions it represents).

Sampling is common in clustering for datasets which are too large to fit in main memory [25, 26]. After clustering the dataset sample, we have a set of clusters with centroids. We then make a single pass through the unclustered intervals and assign each to the cluster that has the nearest center (centroid) to that interval. This then represents the final clustering

Program	# Vecs × # B.B.	SP2	SP3-All	SP3-BinS
gcc-166	4692 × 102038	41 min	9 min	3.5 min
crafty	19189 × 16970	577 min	84 min	10.7 min

Table 2: This table shows the running times (in minutes) by SimPoint 2.0 (SP2), SimPoint 3.0 without using binary search (SP3-All) and SimPoint 3.0 using binary search (SP3-BinS). SimPoint is run searching for the best clustering from k=1 to 100, uses 5 random seeds, and projects the vectors to 15 dimensions. The second column shows how many vectors and the size of the vector (static basic blocks) the programs have.

from which the simulation points are chosen. We originally examined using sub-sampling for variable length intervals in [16]. When using VLIs we had millions of intervals, and had to sub-sample 10,000 to 100,000 intervals for the clustering to achieve a reasonable running time for SimPoint, while still providing very accurate simulation points.

The experiments shown in Figure 2 show the effects of sub-sampling across all the SPEC 2000 benchmarks using 10 million interval size, 30 clusters, 15 projected dimensions, and sub-sampling sizes that used 1/8, 1/4, 1/2, and all of the vectors in each program. The first two plots show the effects of sub-sampling on the CPI errors and k-means variance, both of which degrade gracefully when smaller samples are used. The average SPEC INT and SPEC FP results are shown.

As shown in the second graph of Figure 2, sub-sampling a program can result in k-means finding a slightly less representative clustering, which results in higher k-means variance and higher CPI errors, on average. Even so, when sub-sampling, we found in some cases that it can reduce the k-means variance and/or CPI error (compared to using all the vectors), because sub-sampling can remove unimportant outliers in the dataset that k-means may be trying to fit. It is interesting to note the difference between floating point and integer programs, as shown in the first two plots. It is not surprising that it is easier to achieve lower CPI errors on floating point programs than on integer programs, as the first plot indicates. In addition, the second plot suggests that floating point programs are also easier to cluster, as we can do quite well even with only small samples. The third plot shows the effect of the number of vectors on the running time of SimPoint. This plot shows the time required to cluster the full run of all of the benchmark/input combinations and the three (1/8, 1/4 and 1/2) sub-sampled runs. In addition, we have fit a logarithmic curve with least-squares to the points to give a rough idea of the growth of the run-time. The main variance in time, when two different datasets with the same number of vectors are clustered, is due to the number of k-means iterations required for the clustering to converge.

4.2.2 Number of Dimensions and Random Projection

Along with the number of vectors, the other most important aspect in the running time of k-means is the number of dimensions used. In [2] we chose to use random linear projection to reduce the dimension of the clustered data for SimPoint, which dramatically reduces

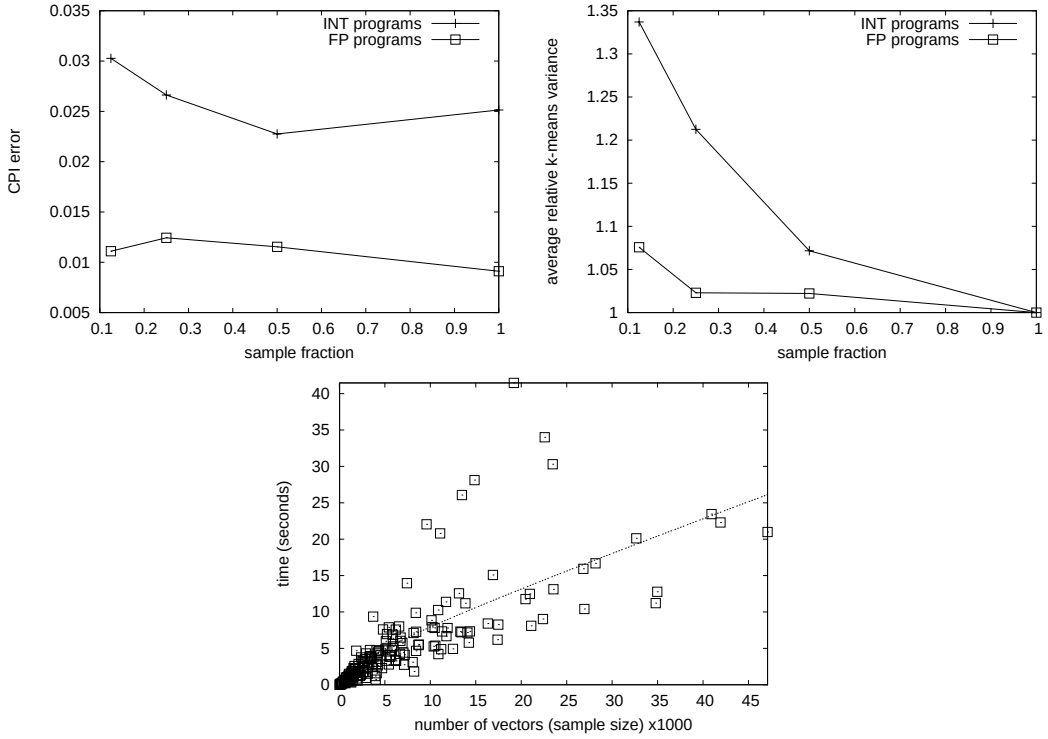

Figure 2: These three plots show how sub-sampling before clustering affects the CPI errors, k-means variance, and the run-time of SimPoint. The first plot shows the average CPI error across the integer and floating-point SPEC benchmarks. The second plot shows the average k-means clustering variance relative to clustering with all the vectors. The last plot shows a scatter plot of the run-time to cluster the full benchmarks and sub-sampled runs, and a logarithmic curve fit with least squares.

computational requirements while retaining the essential similarity information. SimPoint allows the user to define the number of dimensions to project down to. We have found that SimPoint's default of 15 dimensions is adequate for SPEC 2000 applications as shown in [2]. In that earlier work we looked at how much information or structure of frequency vector data is preserved when projecting it down to varying dimensions. We did this by observing how many clusters were present in the low-dimensional version. We noted that at 15 dimensions, we were able to find most of the structure present in the data, but going to even lower dimensions removed too much structure.

To examine random projection, Figure 3 shows the effect of changing the number of projected dimensions on both the CPI error (left) and the run-time of SimPoint (right). For this experiment, we varied the number of projected dimensions from 1 to 100. As the number of dimensions increases, the time to cluster the vectors increases linearly, which is expected. Note that the run-time also increases for very low dimensions, because the points are more "crowded" and as a result k-means requires more iterations to converge.

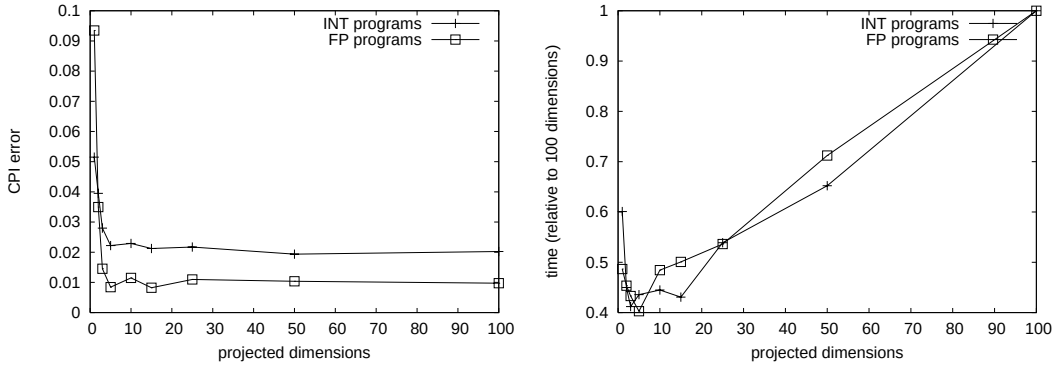

Figure 3: These two plots show the effects of changing the number of projected dimensions when using SimPoint. The default number of projected dimensions SimPoint uses is 15, but here we show results for 1 to 100 dimensions. The left plot shows the average CPI error, and the right plot shows the average time relative to 100 dimensions. Both plots are averaged over all the SPEC 2000 benchmarks, for a fixed $k = 30$ clusters.

It is expected that by using too few dimensions, not enough information is retained to accurately cluster the data. This is reflected by the fact that the CPI errors increase rapidly for very low dimensions. However, we can see that at 15 dimensions, the SimPoint default, the CPI error is quite low, and using a higher number of dimensions does not improve it significantly and requires more computation. Using too many dimensions is also a problem in light of the well-known "curse of dimensionality" [27], which implies that as the number of dimensions increase, the number of vectors that would be required to densely populate that space grows exponentially. This means that higher dimensionality makes it more likely that a clustering algorithm will converge to a poor solution. Therefore, it is wise to choose a dimension that is low enough to allow a tight clustering, but not so low that important information is lost.

4.2.3 Number of Iterations Needed

The final aspect we examine for affecting the running time of the k-means algorithm is the number of iterations it takes for a run to converge.

The k-means algorithm iterates either until it hits a user-specified maximum number of iterations, or until it reaches a point where no further improvement is possible, whichever is less. k-means is guaranteed to converge, and this is determined when the centroids no longer change. In SimPoint, the default limit is 100 iterations, but this can easily be changed (and can be turned off). More than 100 iterations may be required, especially if the number of intervals is very large compared to the number of clusters. The interaction between the number of intervals and the number of iterations required is the reason for the large SimPoint running time for `crafty-ref` in Table 2.

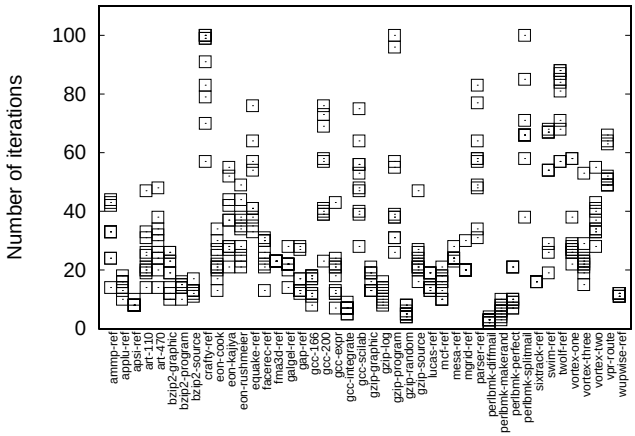

Figure 4: This plot shows the number of iterations required for 10 randomized initializations of each benchmark, with 10-million interval vectors, $k = 30$, and 15 dimensions. Note that only three program/inputs had a total of 5 runs that required more than the default limit of 100 iterations, and these all converge within 160 iterations or less.

For our results, we observed that only 1.1% of all runs on all SPEC 2000 benchmarks reach the limit of 100 iterations. This experiment was with 10-million instruction intervals, k=30, 15 dimensions, and with 10 random (seeds) initializations (runs) of k-means. Figure 4 shows the number of iterations required for all runs in this experiment. Out of all of the SPEC program and input combinations run, only `crafty-ref`, `gzip-program`, `perlbmk-splitmail` had runs that had not converged by 100 iterations. The longest-running clusterings for these programs reached convergence in 160, 126, and 101 iterations, respectively.

4.3 MaxK and Controlling the Number of Simulation Points

The number of simulation points that SimPoint chooses has a direct effect on the simulation time that will be required for those points. The maximum number of clusters, `MaxK`, along with the interval size as discussed in Section 4.1, represents the maximum amount of simulation time that will be needed. When fixed length intervals are used, $MaxK * interval\_size$ puts a limit on the instructions simulated.

SimPoint enables users to trade off simulation time with accuracy. Researchers in architecture tend to want to keep simulation time to below a fixed number of instructions (e.g., 300 million) for a run. If this is desirable, we find that an interval size of 10M with `MaxK=30` provides very good accuracy (as we show in this paper) with reasonable simulation time (below 300 million and around 220 million instructions on average). If even more accuracy is desired, then decreasing the interval size to 1 million and setting `MaxK=300` or `MaxK` equal to the square root of the total number of intervals: $\sqrt{n}$ performs well. Empirically we

discovered that as the granularity becomes finer, the number of phases discovered increases at a sub-linear rate. The upper bound defined by this heuristic works well for the SPEC 2000 benchmarks.

Finally, if the only thing that matters to a user is accuracy, then if SimPoint chooses a number of clusters that is close to the maximum allowed, then it is possible that the maximum is too small to capture all of the unique behaviors. If this is the case and more simulation time is acceptable, it is better to double the maximum k and re-run the SimPoint analysis.

4.3.1 Choosing Simulation Points to Represent the Top Percent of Execution

One advantage to using SimPoint analysis is that each simulation point has an associated weight, which tells how much of the original program's execution is represented by the cluster that simulation point represents. The simulation points can then be ranked in order of importance. If simulation time is too costly, a user may not want to simulate simulation points that have very small weights. SimPoint 3.0 allows the user to specify this explicitly with the `-coveragePct p` option. When this option is specified, the value of `p` sets a threshold for how much of the execution should be represented by the simulation points that are reported in an extra set of files for the simulation points and weights. The default is `p = 1.0`: that the entire execution should be represented.

For example, if `p = 0.98` and the user has specified `-saveSimpoints` and `-saveWeights`, then SimPoint will report simulation points and associated weights for *all* the non-empty clusters in two files, and *also* for the largest clusters which make up at least 98% of the program's weight. Using this reduced-coverage set of simulation points can potentially save a lot of simulation time if there are many simulation points with very small weights without severely affecting the accuracy of the analysis.

Figure 5 shows the effect of varying the percentage of coverage that SimPoint reports. These experiments use binary search with `MaxK=30`, 15 dimensions, and 5 random seeds. The left graph shows the CPI error and the right shows the number of simulation points chosen when only representing the top 95%, 98%, 99% and 100% of execution. The three bars show the maximum value, the second highest value (max-1), and the average. The results show that when the coverage is reduced from 100%, the average number of simulation points decreases, which reduces the simulation time required, but this is at the expense of the CPI error, which goes up on average. For example, comparing 100% coverage to 95%, the average number of simulation points is reduced from about 22 to about 16, which is a reduction of about 36% in required simulation time for fixed-length vectors. At the same time, the average CPI error increases from 1.5% to 2.8%. Depending on the user's goal, a practitioner can use these types of results to decide on the appropriate trade off between simulation time and accuracy. Out of all of the SPEC binary/input pairs there was one combination (represented by the maximum) that had a bad error rate for 95% and 98%. This was `ammp-ref`, and the reason was that a simulation point was removed that had a small weight (1-2% of the executed instructions) but its behavior was different enough to affect the estimated CPI.

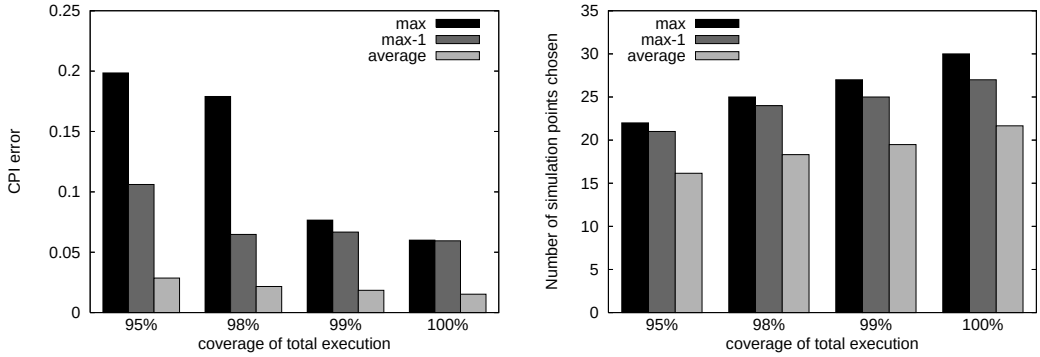

Figure 5: These plots show the CPI error and number of simulation points picked across different percent coverages of execution. For 100% coverage, all simulation points are used, but for less than 100%, simulation points from the smallest clusters are discarded, keeping enough simulation points to represent the desired coverage. Bars labeled "max-1" show the second largest value observed.

Note, when using simulation points for an architecture design space exploration, the CPI error compared to the baseline is not as important as making sure that this error is consistent between the different architectures being examined. What is important is that a consistent relative error is seen across the design space exploration, and SimPoint has this consistent bias as shown in [3]. Ignoring a few simulation points using `-coveragePct p` will create a consistent bias across the different architecture runs when compared to complete simulation. This is because a small fraction of behavior will be ignored during the design space exploration, but the same simulation points representing the top percent of execution will be represented. This can be acceptable technique for reducing simulation time, especially when performing large design space exploration trade-offs.

4.4 Searching for the Smallest k with Good Clustering

As described above, we suggest setting `MaxK` as appropriate for the maximum amount of simulation time a user will tolerate for a given run. We then use three techniques to search over the possible values of k, which we describe now. The goal is to try to pick a k that reduces simulation time, but also provides an accurate picture of the program's execution.

4.4.1 Setting the BIC Percentage

As we examine several clusterings and values of k, we need to have a method for choosing the best clustering. The *Bayesian Information Criterion* (BIC) [19] gives a score of the goodness of the clustering of a set of data. These BIC scores can then be used to compare different clusterings of the same data. The BIC score is a penalized likelihood of the clustering of the vectors, and can be considered the approximation of a probability. However, the BIC score often increases as the number of clusters increase. Thus choosing the clustering with

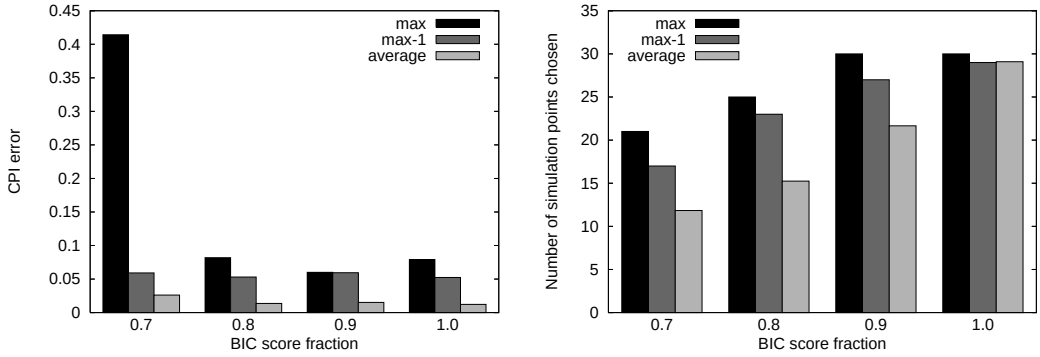

Figure 6: These plots show how the CPI error and number of simulation points chosen is affected by varying the BIC threshold. Bars labeled "max-1" show the second largest value observed.

the highest BIC score can lead to often selecting the clustering with the most clusters. Therefore, we look at the range of BIC scores, and select the score which attains some high percentage of this range. The SimPoint default BIC threshold is 0.9. When the BIC rises and then levels off, this method chooses a clustering with the fewest clusters that is near the maximum value. Choosing a lower BIC percent would prefer fewer clusters, but at the risk of less accurate simulation.

Figure 6 shows the effect of changing the BIC threshold on both the CPI error (left) and the number of simulation points chosen (right). These experiments are for using binary search with `MaxK=30`, 15 dimensions, and 5 random seeds. BIC thresholds of 70%, 80%, 90% and 100% are examined. As the BIC threshold decreases, the average number of simulation points decreases, and similarly the average CPI error increases. At the 70% BIC threshold, `perlbmk-splitmail` has the maximum CPI error in the SPEC suite. This is due to a clustering that was picked at that threshold which has only 9 clusters. This anomaly is an artifact of the looser threshold, and better BIC scores point to better clusterings and better error rates, which is why we recommend the BIC threshold to be set at 90%.

4.4.2 Varying the Number of Random Seeds, and k-means initialization

The k-means clustering algorithm is essentially a hill-climbing algorithm, which starts from a randomized initialization, which requires a random seed. Because of this, running k-means multiple times can produce very different results depending on the initializations. Sometimes this means k-means can converge to a locally-good solution that is poor compared to the best clustering on the same data for that number of clusters. Therefore the conventional suggests that it is good to run k-means several times using a different randomized starting point each time, and take the best clustering observed, based on the k-means variance or the BIC. SimPoint has the functionality to do this, using different random seeds to initialize

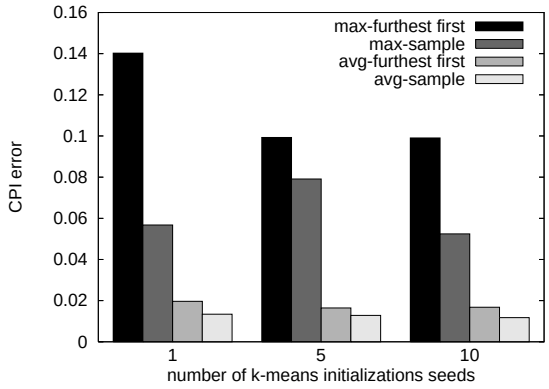

Figure 7: This plot shows the average and maximum CPI errors for both sampling and furthest-first k-means initializations, and using 1, 5, or 10 different random seeds. These results are over the SPEC 2000 benchmark suite for 10-million instruction vectors, 15 dimensions, and $k = 30$.

k-means each time. Based on our experience, we have found that using 5 random seeds works well.

Figure 7 shows the effect on CPI error of using two different k-means initialization methods (furthest-first and sampling) along with different numbers of initial k-means seeds. These experiments are for using binary search with `MaxK=30`, 15 dimensions, and a BIC threshold of .9. When multiple seeds are used, SimPoint runs k-means multiple times with different starting conditions and takes the best result.

Based on these results we see that sampling outperforms furthest-first k-means initialization. This can be attributed to the data we are clustering, which has a large number of anomaly points. The furthest-first method is likely to pick those anomaly points as initial centers since they are the furthest points apart. The sampling method randomly picks points, which on average does better than the furthest-first method. It is also important to try multiple seed initializations in order to avoid a locally minimal solution. The results in Figure 7 shows that 5 seed initializations is sufficient for finding a good clustering, but using 10 seeds did reduce the maximum error seen from 8% down to 5.5%.

4.4.3 Binary Search for Picking k

SimPoint 3.0 makes it much faster to find the best clustering and simulation points for a program trace over earlier versions. Since the BIC score generally increases as k increases, SimPoint 3.0 uses this to perform a binary search for the best k. For example, if the maximum k desired is 100, with earlier versions of SimPoint one might search in increments of 5: $k = 5, 10, 15, \ldots, 90, 100$, requiring 20 clusterings. With the binary search method, we can ignore large parts of the set of possible k values and examine only about 7 clusterings.

The binary search method first clusters 3 times: at $k = 1$, $k = \max \mathrm{k}$, and $k = (\max \mathrm{k} + 1)/2$. It then proceeds to divide the search space and cluster again based on the

BIC scores observed for each clustering. The binary search may stop early if the window of k values is relatively small compared to the maximum k value. Thus the binary search method requires the user only to specify the maximum k value, and performs at most log(max k) clusterings.

Figure 8 shows the comparison between the new binary search method for choosing the best clustering, and searching all k values (as was done in SimPoint 2.0). The top graph shows the CPI error for each program, and the bottom graph shows the number of simulation points (clusters) chosen. These experiments are for using binary search with `MaxK=30`, 15 dimensions, 5 random seeds, and a BIC threshold of .9. SimPoint All performs slightly better than the binary search method, since it searches exhaustively through all k values for `MaxK=30`. Using the binary search, it is possible that it will not choose as small of clustering as the exhaustive search. This is shown in the bottom graph of Figure 8, where the exhaustive search picked 19 simulation points on average, and binary search chose 22 simulation points on average. In terms of CPI error rates, the average is about the same across the SPEC programs between exhaustive and binary search.

5. Common Pitfalls

There are a few important potential pitfalls worth addressing to ensure accurate use of SimPoint's simulation points.

Setting MaxK Appropriately – MaxK must be set based on the interval size used and the maximum number of instructions you are willing to simulate as described in Section 4.3.

The maximum number of clusters and the interval size represent the maximum amount of simulation time needed for the simulation points selected by SimPoint. Finding good simulation points with SimPoint requires recognizing the tradeoff between accuracy and simulation time. If a user wants to place a low limit on the number of clusters to limit simulation time, SimPoint can still provide accurate results, but some intervals with differing behaviors may be grouped together as a result. In such cases it may be advantageous to increase $MaxK$ and with that use the option `-coveragePct` with a value less than 1 (e.g. .98). This can allow different behaviors to be grouped into more clusters, but the final set of simulation points can be smaller since only the most dominant behaviors will be chosen for simulation points.

Off by One Interval Errors – SimPoint 3.0 starts counting intervals and cluster IDs at 0. These are the counts and IDs written to a file by `-saveSimpoints`, where SimPoint indicates which intervals have been selected as simulation points and their respective cluster IDs. A common mistake may be to assume that SimPoint 3.0, like previous versions of SimPoint, counts intervals starting from 1, instead of 0. Just remember that the first interval of execution and the first cluster in SimPoint 3.0 are both numbered 0.

Reproducible Tracking of Intervals and Using Simulation Points – It is very important to have a reproducible simulation environment for (a) creating interval vectors, and (b) using the simulation points during simulation. If the instruction counts are not stable between runs, then selection of intervals can be skewed, resulting in additional error.

SimPoint provides the interval number for each simulation point. Interval numbers are zero-based, and are relative to the start of execution, not to the previous simulation

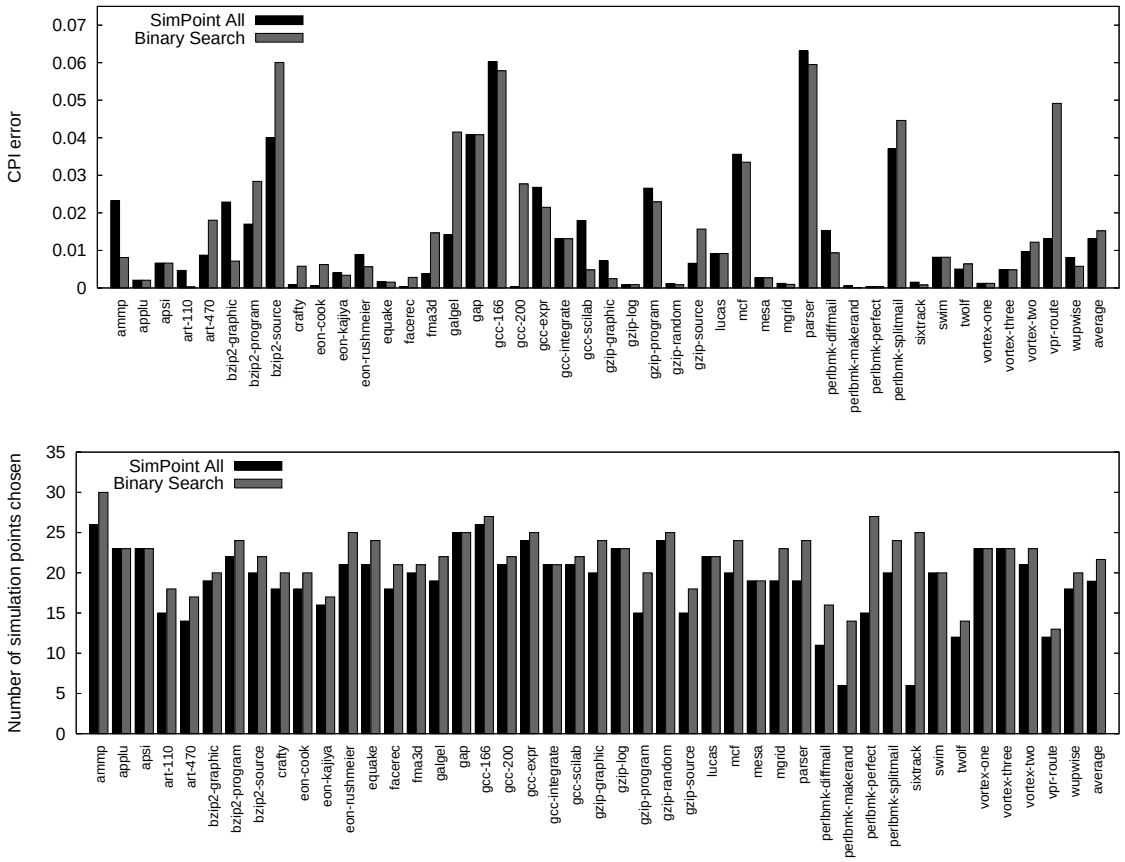

Figure 8: These plots show the CPI error and number of simulation points chosen for two different ways of searching for the best clustering. The first method, which was used in SimPoint 2.0, is searching for All k between 1 and 30, and choosing the smallest clustering that achieves the BIC threshold. The second method is the binary search for `MaxK=30`, which examines at most 5 clusterings.

point. So for fixed-length intervals, to get the instruction count at the start of a simulation point, just multiply the interval number by the interval size, but watch out for Interval Drift described later. For example, interval number 15 with an interval size of 10 million instructions means that the simulation point starts when 150 million (15*10M) correct path instructions have been fetched. Detailed simulation of this simulation point would occur from instruction 150 million until just before 160 million.

One way to get more reproducible results is to use the first instruction program counter (Start PC) that occurs at the start of each interval of execution, instead of relying on instruction count. The same program counter can reappear many times, so it is also neces-

sary to keep track of how many times a program counter value must appear to indicate the start of an interval of execution. For example, if a simulation point is triggered when PC 0x12000340 is executed the 1000th time. Then detailed simulation starts after that PC is seen 1000 times, and simulation occurs for the length of the interval. For this to work, the user needs to profile PCs in parallel with the frequency vector profile, and record the first PC seen for each interval along with the number of times that PC has executed up to that point in the execution. SimPoint provides the interval chosen for a simulation point, and this data can easily be mapped to this PC profile to determine the Start PC and the Nth occurrence of it where simulation should start.

It is highly recommended that you use the simulation point Start PCs for performing simulations. There are two reasons for this. The first reason deals with making sure you calculate the instructions during fast-forwarding exactly the same as when the simulation points were gathered. The second reason is that there can be slight variations in execution count between different runs of the same binary/input due to subtle changes in the simulation environment. Both of these are discussed in more detail later in this section.

Note, if you use the Start PC and its invocation count you need to make sure that the binary and any shared libraries used are loaded into the same address locations across all of your simulation runs for this to work. In general, this is important for any simulation study, in order to ensure that there are consistent address streams (instruction and global data) seen across the different runs of a program/input pair.

Interval "Drift" – When creating intervals, a problem may occur that the counts inside an interval might be just slightly larger than the interval size. Over time these counts can add up, so that if you were to try to find a particular fixed length interval in a simulation environment different from where the intervals were generated, you might be off by a few intervals.

For example, this can occur when forming fixed length intervals of X instructions. After X instructions execute the interval should be created, but this boundary may occur in the middle of a basic block, where there are an additional Y instructions in the basic block over the interval size. A frequency vector profiler that has the problem of interval drift may naively include these additional Y instructions in the interval that was just completed, especially if it was just counting basic blocks. Even though Y may be extremely small, it will accumulate over many thousands of intervals and cause a slow "drift" in the interval endpoints in terms of instruction count.

This is mainly a problem if you use executed instructions to determine the starting location for a simulation point. If you have drift in your intervals, to calculate the starting instruction count, you cannot just multiply the simulation point by the fixed length interval size as described above, since the interval lengths are not exactly the same. This can result in simulating the wrong set of instructions for the simulation point. When using the instruction count for the start of the simulation point, you need to keep track of the total instruction count for each interval if you have interval drift. You can then calculate the instruction count starting location for a simulation point by summing up the exact instruction counts for all of the intervals up to the interval chosen as the simulation point.

A better solution is to just make sure there is no drift at all in your intervals, but ending them precisely at the interval size boundary. In our above example, instead of including Y extra instructions in the interval that just ended, those extra Y instructions should be

counted toward their basic block in the next interval. This results in splitting the basic block counts, for the basic blocks that occur on an interval boundary.

Accurate Instruction Counts (No-ops) – It is important to count instructions exactly the same for the frequency vector profiles as for the detailed simulation, otherwise they will diverge. Note that the simulation points on the SimPoint website include only correct path instructions and the instruction counts include no-ops. Therefore, to reach these simulation points in a simulator, *every* committed (correct path) instruction (including no-ops) must be counted.

System Call Effects – Some users have reported system call effects when running the same simulation points under slightly different OS configurations on a cluster of machines. This can result in slightly more or fewer instructions being executed to get to the same point in the program's execution, and if the number of instructions executed is used to find the simulation point, this may lead to variations in the results. To avoid this, we suggest using the Start PC and Execution Count for each simulation point as described above. Another way to avoid variations in startup is to use checkpointing [22], and to use the SimpleScalar EIO files to make sure the system calls are the same between all simulated runs of a program/input combinations.

Calculating Weighted IPC – For IPC (instructions/cycle) we cannot just apply the weights directly as is done for CPI. Instead, we must convert all the simulated samples to CPI, compute the weighted average of CPI, and then convert the result back to IPC.

Calculating Weighted Miss Rates – To compute an overall miss rate (e.g. cache miss rate), first we must calculate both the weighted average of the number of cache accesses, and the weighted average of the number of cache misses. Dividing the second number by the first gives the estimated cache miss rate. In general, care must be taken when dealing with any ratio because both the numerator and the denominator must be averaged separately and *then* divided.

Number of intervals – There should be a sufficient number of intervals for the clustering algorithm to choose from. A good rule of thumb is to make sure to use at least 1,000 intervals in order for the clustering algorithm to be able to find a good partition of the intervals. If there are too few intervals, one can decrease the interval size to obtain more intervals for clustering.

Using SimPoint 2.0 with VLIs – As described in Section 4.1.1, SimPoint 2.0 assumes fixed-length intervals, and should not be used if the vectors to be clustered are variable length. The problem with using VLIs with SimPoint 2.0 is that the data will be clustered with a uniform weight distribution across all intervals, which is not correct for representing the execution properly. This means that the centroids may not be representative of the program's execution in a cluster. This can result in large error rates, since a vector that is not representative of the majority of the cluster could be chosen as the simulation point.

Wanting Variable Length, but not asking for it – If you want variable length weighting for each interval then you need to use the `-fixedLength off` option. You may need to also use `-loadVectorWeights` if your vector weights cannot be automatically calculated from the vector's frequency count values.

6. Summary

Modern computer architecture research depends on understanding the cycle level behavior of a processor running an application, and gaining this understanding can be done efficiently by judiciously applying detailed cycle level simulation to only a few simulation points. The level of detail provided by cycle level simulation comes at the cost of simulation speed, but by targeting only one or a few carefully chosen samples for each of the small number of behaviors found in real programs, this cost can be reduced to a reasonable level.

The main idea behind SimPoint is the realization that programs typically only exhibit a few unique behaviors which are interleaved with one another through time. By finding these behaviors and then determining the relative importance of each one, we can maintain both a high level picture of the program's execution and at the same time quantify the cycle level interaction between the application and the architecture. The key to being able to find these phases in a efficient and robust manner is the development of a metric that can capture the underlying shifts in a program's execution that result in the changes in observed behavior. SimPoint uses frequency vectors to calculate code similarity to cluster a program's execution into phases.

SimPoint 3.0 automates the process of picking simulation points using an off-line phase classification algorithm, which significantly reduces the amount of simulation time required. These goals are met by simulating only a handful of *intelligently* picked behaviors of the full program. When these simulation points are carefully chosen, they provide an accurate picture of the complete execution of a program, which gives a highly accurate estimation of performance. This release provides new features for reducing the run-time of SimPoint and simulation points required, and provides support for variable length intervals. The SimPoint software can be downloaded at:

`http://www.cse.ucsd.edu/users/calder/simpoint/`

Acknowledgments

This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft.

Appendix A: Command Line Options

Clustering and projection options:

- `-k regex`: This specifies which values of k should be searched. The regular expression is

```
regex := "search" | R(,R)*
    R := k | start:end | start:step:end
```

  `Search` means that SimPoint should search using a binary search between 1 and the user-specified `maxK`. The `-maxK` option must be set for `search`. Searching is the default behavior. If the user chooses not to use `search`, they may specify one or more comma-separated ranges of positive integers for k. The argument `k` specifies a single k value,

the range `start:end` indicates that all integers from `start` to `end` (inclusive) should be used, and the range `start:step:end` indicates that SimPoint should use values starting at `start` and stepping by interval `step` until reaching `end`. Here is an example of specifying specific values with the regular expression: `-k 4:6,10,12,30:15:75`, which represents searching the k values 4,5,6,10,12,30,45,60,75.

- `-maxK k`: When using the "search" clustering method (see `-k` option), this command line option must be provided. It specifies the maximum number of clusters that SimPoint should use.

- `-fixedLength "on" | "off"`: Specifies whether the frequency vectors that are loaded should be treated as fixed-length vectors (which means having equal weights), or VLI vectors. The default is on. When off, if no weights are loaded (using `-loadVectorWeights`) then the weight of each interval is determined by summing up all the frequency counts in the vector for an interval and dividing this by the total frequency count over all intervals.

- `-bicThreshold t`: SimPoint finds the highest and lowest BIC scores for all examined clusterings, and then chooses the one with the smallest k which has a BIC score greater than `t*(max_score-min_score)+min_score`. The default value for `t` is 0.9.

- `-dim d | "noProject"`: `d` is the number of dimensions down to which SimPoint should randomly project the un-projected frequency vectors. If the string "noProject" is instead given, then no projection will be done on the data. If the `-dim` option is not specified at all, then a default is 15 dimensions is used. This option does not apply when loading data from a pre-projected vector file using options `-loadVectorsTxtFmt` or `-loadVectorsBinFmt`.

- `-seedproj seed`: The random number seed for random projection. The default is 2042712918. This can be changed to any integer for different random projections.

- `-initkm "samp" | "ff"`: The type of k-means initialization (sampling or furthest-first). The default is `"samp"`. Sampling chooses k different vectors from the program at random as the initial cluster centers. Furthest-first chooses a random vector as the first cluster center, then repeats the following $k-1$ times: find the closest center to each vector, and choose as the next new center the vector which is furthest from its closest center.

- `-seedkm seed`: The random number seed for k-means initialization (see `-initkm`). The default is 493575226. This can be changed to any integer to obtain different k-means initializations, and using the same seed across runs will provide reproducible initializations.

- `-numInitSeeds n`: The number of random initializations to try for clustering each k. For each k, the dataset is clustered `num` times using different k-means initializations (the k-means initialization seed is changed for each initialization). Of all the `num` runs, only the best (the one with the highest BIC score) is kept. The default is 5.

- `-iters n | "off"`: The maximum number of k-means iterations per clustering. The default is 100, but the algorithm often converges and stops much earlier. If "off" is instead chosen, then k-means will terminate once it has converged. In running all of the SPEC programs with all of their inputs using the default parameters to SimPoint 3.0 only 1.1% of all runs did not converge by 100 iterations. Clearly, the default number of iterations is usually sufficient, but can be increased if SimPoint is often reaching the limit.

- `-verbose level`: The amount of output that SimPoint should produce. The argument `level` is a non-negative integer, where larger values indicate more output. The default is 0, which is the minimum amount of output.

Sampling options:

- `-sampleSize n`: The *number* of frequency vectors (intervals) to randomly sample before clustering with k-means. Using a smaller number of vectors allows k-means to run faster, at a small cost in accuracy. The vectors are sampled without replacement, so each vector can be sampled only once. For VLI vectors, vectors are chosen with probability proportional to how much of the execution they represent. The default is to use all vectors for clustering.

- `-seedsample seed`: The random number seed for vector sampling. The default is 385089224. This can be changed to any integer for different samples.

Load options:

- `-loadFVFile file`: Specifies an unprojected sparse-format frequency vector (FV) file of all of the intervals. Either this argument, `-loadVectorsTxtFmt`, or `-loadVectorsBinFmt` must always be present to provide SimPoint with the frequency vectors that should be analyzed.

- `-numFVs n`, `-FVDim n`: These two options together specify the number of frequency vectors and maximum number of dimensions in the unprojected frequency vector file so the file doesn't need to be parsed twice (both options must be used together).

- `-loadVectorsTxtFmt file`: Specifies an already-projected *text* vector file (saved with `-saveVectorsTxtFmt`). When loaded this way, SimPoint does not use random projection or otherwise change the vectors.

- `-loadVectorsBinFmt file`: Specifies an already-projected *binary* vector file (saved with `-saveVectorsBinFmt`). This is the binary-format version of `-loadVectorsTxtFmt`. This option provides the fastest way to load a dataset.

- `-inputVectorsGzipped`: When present, this option specifies that the input vectors given by `-loadFVFile`, `-loadVectorsTxtFmt`, or `-loadVectorsBinFmt` are compressed with gzip compression, and should be decompressed while reading.

Option	Default Value
-k	"search"
-initkm	"samp"
-numInitSeeds	5
-bicThreshold	0.9
-fixedLength	"on"
-dim	15
-iters	100
-sampleSize	no sub-sampling
-coveragePct	1 (100%)

Table 3: This table gives the standard options that are used with SimPoint and their default values. For every run of SimPoint, the frequency vectors must be provided as an unprojected frequency vector file, or a pre-projected data file given via `-loadVectorsTxtFmt` or `-loadVectorsBinFmt`. When using the `-k "search"` method, `-maxK` must always be provided.

- `-loadInitCtrs file`: Specifies initial centers for clustering (rather than allowing SimPoint to choose the initial centers with furthest-first or sampling). These centers are points in the same dimension as the projected frequency vectors, but they are not necessarily actual frequency vectors. This option is incompatible with using multiple values of k; only the k corresponding to the number of centers in the given file will be run. This is useful if you want to specify the exact starting centers to perform a clustering.
- `-loadInitLabels file`: Specifies the labels that will be used to form initial clusters (rather than allowing SimPoint to choose with furthest-first or sampling). Like `-loadInitCtrs`, this option is incompatible with multiple k values. This is used if you want to specify the initial starting clusters to perform a clustering based on a set of labels. In doing this, the new starting centers will be formed from these labels and clustering iterations will proceed from there.
- `-loadProjMatrixTxtFmt file`: Specifies a *text* projection matrix to use to project the unprojected frequency vector file (saved from a previous run with `-saveProjMatrixTxtFmt`), rather than allowing SimPoint to choose a random projection matrix. This option also allows users to specify their own projection matrix.
- `-loadProjMatrixBinFmt file`: Specifies a *binary* projection matrix to use to project the unprojected frequency vector file. This is the binary version of `-loadProjMatrixTxtFmt`.
- `-loadVectorWeights file`: Specifies a text file that contains the weights that should be applied to the frequency vectors. The weights should all be non-negative, and their sum should be positive.

Save options:

- `-saveSimpoints file`: Saves a file of the vectors chosen as Simulation Points and their corresponding cluster numbers. Frequency vectors are numbered starting at 0, which means the first vector in the execution has an index of 0. *Note* that earlier versions of SimPoint started numbering vectors from 1.

- `-saveSimpointWeights file`: Saves a file containing a weight for each Simulation Point, and its corresponding cluster number. The weight is the proportion of the program's execution that the Simulation Point represents.

- `-saveVectorWeights file`: Saves a file with a weight for each frequency vector as computed by SimPoint. The weight of a vector is the proportion that vector represents of the all of the vectors provided. When using VLIs (and the option `-fixedLength off`, this is calculated for a vector by taking the total value of all of the entries in a vector divided by the total value of all of the entries in all vectors. The weights are also stored in projected vector files saved with `-saveVectorsTxtFmt` and `-saveVectorsBinFmt`, so this option is not necessary for just saving and loading projected data.

- `-saveAll`: When this option is `not` specified, SimPoint only saves specified outputs for the best clustering found (according to the BIC threshold). When this option is specified, SimPoint will save the specified outputs for all k values clustered. This option only affects saving labels, simulation point weights, simulation points, initial centers, and final centers.

- `-coveragePct p`: This option tells SimPoint to save additional simulation points and weights that belong to the largest clusters that together make up at least `p` proportion of the vector weights for the entire program. The range of `p` is between 0 and 1; the default is 1. For example, .98 means to output the smallest number of simulation points to account for at least 98% of execution (vectors). This option only affects the saving of simulation points and simulation point weights as a special coverage percentage result. The simulation points and associated weights for all clusters will also be saved as usual.

- `-saveVectorsTxtFmt file`: Specifies the file in which to save a text version of the projected frequency vectors to enable faster loading later. See `-loadVectorsTxtFmt`.

- `-saveVectorsBinFmt file`: Specifies the file in which to save a binary version of the projected frequency vectors to enable faster loading later. See `-loadVectorsBinFmt`.

- `-saveProjMatrixTxtFmt file`: Specifies the file in which to save a text version of the projection matrix so it may be re-used. See `-loadProjMatrixTxtFmt`.

- `-saveProjMatrixBinFmt file`: Specifies the file in which to save a binary version of the projection matrix so it may be re-used. See `-loadProjMatrixBinFmt`.

- `-saveInitCtrs file`: Specifies the file in which to save the initial cluster centers.

- `-saveFinalCtrs file`: Specifies the file in which to save the final cluster centers found by k-means.

- `-saveLabels file`: Specifies the file in which to save the final label and distance from cluster center for each clustered vector.

Table 3 shows all of the default values and required options for running SimPoint. The two required parameters for every run of SimPoint are providing the frequency vectors (one of `-loadFVFile`, `-loadVectorsTxtFmt`, or `-loadVectorsBinFmt`) and the range of k values using either `-k` or `-maxK`.

References

[1] 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*, Sept. 2001.

[2] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *10th International Conference on Architectural Support for Programming*, Oct. 2002.

[3] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[4] M. V. Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2004.

[5] J. Lau, S. Schoenmackers, and B. Calder, "Structures for phase classification," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2004.

[6] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder, "The strong correlation between code signatures and performance," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

[7] P. Denning and S. C. Schwartz, "Properties of the working-set model," *Communications of the ACM*, vol. 15, pp. 191–198, Mar. 1972.

[8] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *33rd International Symposium on Microarchitecture*, pp. 245–257, 2000.

[9] A. Dhodapkar and J. E. Smith, "Dynamic microarchitecture adaptation via co-designed virtual machines," in *International Solid State Circuits Conference*, Feb. 2002.

[10] A. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *29th Annual International Symposium on Computer Architecture*, May 2002.

[11] A. Dhodapkar and J. Smith, "Comparing program phase detection techniques," in *36th Annual International Symposium on Microarchitecture*, Dec. 2003.

[12] M. Hind, V. Rjan, and P. Sweeney, "Phase shift detection: A problem classification," tech. rep., IBM, Aug. 2003.

[13] C. Isci and M. Martonosi, "Identifying program power phase behavior using power vectors," in *Workshop on Workload Characterization*, Sept. 2003.

[14] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *36th International Symposium on Microarchitecture*, Dec. 2003.

[15] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *12th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2003.

[16] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

[17] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *IEEE International Symposium on Workload Characterization*, Oct. 2005.

[18] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (L. M. LeCam and J. Neyman, eds.), vol. 1, (Berkeley, CA), pp. 281–297, University of California Press, 1967.

[19] 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*, pp. 727–734, 2000.

[20] 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.

[21] G. Hamerly, E. Perelman, and B. Calder, "How to use SimPoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, Mar. 2004.

[22] M. V. Biesbrouck, L. Eeckhout, and B. Calder, "Efficient sampling startup for sampled processor simulation," in *International Conference on High Performance Embedded Architectures and Compilers*, Nov. 2005.

[23] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *International Symposium on Microarchitecture*, Dec. 2004.

[24] J. Lau, E. Perelman, and B. Calder, "Selecting software phase markers with code structure analysis," Tech. Rep. UCSD-CS2004-0804, UC San Diego, Nov. 2004.

[25] F. Farnstrom, J. Lewis, and C. Elkan, "Scalability for clustering algorithms revisited," *SIGKDD Explor. Newsl.*, vol. 2, no. 1, pp. 51–57, 2000.

[26] F. J. Provost and V. Kolluri, "A survey of methods for scaling up inductive algorithms," *Data Mining and Knowledge Discovery*, vol. 3, no. 2, pp. 131–169, 1999.

[27] R. E. Bellman, *Adaptive Control Processes*. Princeton University Press, 1961.