How to Use SimPoint to Pick Simulation Points

Greg Hamerly Erez Perelman Brad Calder

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

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 [11, 12, 10, 1, 7]. 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. The SimPoint software can be downloaded at:

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

SimPoint is one specific use of our off-line phase clustering approach [12] motivated by the need to perform efficient and accurate program analysis and architecture simulation, and several researchers in academia and at Intel are using SimPoint to accurately guide their architecture simulation research.

2 Automatically Finding Phase Behavior

In this section we provide a brief overview of phase analysis and the clustering approach to group similar parts of a program's execution together.

2.1 Phase Behavior

The way a program behaves over time is not totally random; in fact, it often falls into repeating behaviors, which we call phases. Automatically identifying this phase behavior is the goal of our research presented by Sherwood et.al. [12], and the key to unlocking many new optimizations.

To identify phases, we break a program's execution into contiguous non-overlapping intervals. An *interval* is a continuous portion of execution (a slice in time) of a program. For our studies we have used interval sizes of 1 million, 10 million and 100 million instructions [10], and we are currently examining a variable interval size SimPoint algorithm. A *phase* is a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency. This means that a phase may appear many times as a program executes. *Phase classification* partitions a set of intervals into phases with similar behavior. The phases that we discover are specific to the input used to run the program.

The key observation for phase recognition is that any program metric is a direct function of the way a program traverses the code during execution. We can find this phase behavior and classify it by examining only the ratios in which different regions of code are being executed over time. We can simply and quickly collect this information using code profiles. Accurately capturing phase behavior by only examining program or ISA-level metrics, independent of the underlying architectural details and performance, allows us to choose architecture independent simulation points. This means that it is possible to use phase information to guide many optimizations and policy decisions across different architecture configurations when performing a design space search.

2.2 Profiling Granularity

SimPoint uses the phases generated by the off-line analysis to intelligently choose where to spend simulation time. At the highest level, SimPoint calculates phases for a program/input pair, and then chooses a single representative from each phase. The representative for each phase is chosen by finding the interval most similar to the average behavior of the phase. This representative interval for a phase is called a simulation point for that phase. Users can then perform program analysis or collect statistics on detailed simulation only on these simulation points. The whole program's behavior is then estimated by combining the weighted performance results of each simulation point. The weights come from the size (proportion of the whole program) of the phase it comes from. This methodology allows SimPoint to significantly reduce program analysis and simulation time and provide an accurate characterization of the full program.

2.3 Data Structures Used to Capture Phase Behavior

The first step of our phase analysis is to quickly profile the frequency of the code being executed to create code signatures that represent the program's behavior at different points during execution. These signatures, which are vectors, are then

used along with several techniques from clustering analysis to concisely group together similar parts of the program's execution into phases, or clusters.

2.3.1 Basic Block Frequency Vectors: Our prior approach uses the Basic Block Vector (or BBV) [11] as a structure designed to capture information about changes in a program's behavior over time. A basic block is a single-entry, single-exit section of code with no internal control flow. More formally, a *Basic Block Vector* is a one dimensional array, where each element in the array corresponds to one static basic block in the program. We start with a BBV containing all zeroes at the beginning of each interval. During each interval, we count the number of times each basic block in the program has been entered, and we record the count in the BBV. For example, if the 50th basic block is executed 15 times in an interval, then bbv[50] = 15 for that interval. In addition, we multiply each count by the number of instructions in the basic block, so basic blocks containing more instructions will have more weight in the BBV. Finally, at the end of each interval, we normalize the basic block vector by dividing each element by the sum of all the elements in the vector.

We use BBVs to compare the intervals of the application's execution. 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 [11]. 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 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 performance of those two intervals to be similar.

Recently, we examined frequency vector structures other than basic block vectors for the purpose of phase classification. We have looked at frequency vectors for loops, procedures, register usage, instruction mix, and memory behavior [7]. We found that using register usage vectors, which simply count for a given interval the number of times each register is defined and used, provides similar accuracy to using basic block vectors. In addition, tracking only loop and procedure branch execution frequencies performed almost as well as using the full basic block information.

2.4 Using Clustering for Phase Classification

Frequency vectors provide a compact and representative summary of the program's behavior for each interval of execution. By examining the similarity between them, it is clear that there are high-level patterns in each program's execution.

To exploit phase behavior, it is useful to have an automated way of extracting phase information from programs. To break the complete execution of the program into phases that have similar frequency vectors, clustering algorithms from the field of machine learning have been shown to be very effective [12]. Because the frequency vectors relate to the overall performance of the program, grouping intervals based on their frequency vectors produces phases that are similar not only in the distribution of program structures used, but also in every other architecture metric measured, including overall performance.

The goal of clustering is to divide 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. A well known clustering algorithm is k-means [8], which we use to accurately split program behavior into phases. We use random linear projection [4], which reduces the dimensionality of the input data while preserving the underlying similarity information, to speed up the execution of k-means. One drawback of the k-means algorithm is that it requires the number of clusters k as an input to the algorithm, but we do not know beforehand what value is appropriate. To address this, we run the algorithm for several values of k, and then use a goodness score to guide our final choice for k.

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 a *minimum number of clusters* where each cluster has a certain level of homogeneous behavior.

The following steps summarize the phase clustering algorithm at a high level. We refer the interested reader to [12] for a more detailed description of each step.

1. Profile the program by dividing the program's execution into contiguous intervals of size N (e.g., 1 million, 10 million, or 100 million instructions). For each interval, collect a frequency vector tracking the program's use of some program structure (basic blocks, loops, register usage, etc.). This generates a frequency vector for every 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 D dimensions using random linear projection. The advantage of performing clustering on projected data is that it speeds up the k-means algorithm significantly, and reduces the memory requirements by several orders of magnitude over using the original vectors, while preserving the essential similarity information.

3. Run the k-means clustering algorithm on the reduced dimensional data with values of k from 1 to K, where K (Max K) is the maximum number of phases that can be detected. Each run of k-means produces a clustering, which is a partition of the data into k different phases/clusters. Each run of k-means begins with a random initialization step, which requires a random seed.

4. To compare and evaluate the different clusters formed for different k, we use the *Bayesian Information Criterion* (BIC) [9] as a measure of the "goodness of fit"

of a clustering to a dataset. More formally, the BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering is a good fit to the data. For each clustering ($k = 1 \ldots M$), the fitness of the clustering is scored using the BIC formulation given in [9].

5. The final step is to choose the clustering with the smallest k, such that its BIC score is at least B% as good as the best score. The clustering k chosen is the final grouping of intervals into phases.

The above algorithm groups intervals into phases. We use the Euclidean distance between vectors as our similarity metric. This algorithm has important parameters (N, D, K, B, and more) to tune to create accurate and representative simulation points using SimPoint. We discuss these parameters in detail in Section 5.

3 How SimPoint Chooses Simulation Points

After the phase classification algorithm described in the previous section has done its job, intervals with similar code usage will be grouped together into the same phase, or cluster. Then from each phase, we choose one representative interval that will be simulated in detail to represent the behavior of the whole phase. Therefore, 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 that is closest to the center, or *centroid*, of each cluster. The centroid is the average of all the intervals in the cluster. This is analogous to the center of mass of all the points that are in that cluster. It can also be viewed as the interval which behaves most like the average behavior of the entire phase. Most likely there is no interval that exactly matches the centroid, so the interval closest to the centroid is chosen. The selected interval is called a *Simulation Point* for that phase [10, 12]. Detailed simulation is then performed on the set of simulation points.

SimPoint outputs a weight for each simulation point. Each weight is a fraction: the number of instructions in the cluster from which the simulation point was taken over the number of instructions in the program. With the weights and the detailed simulation results of each simulation point, we compute a weighted average that is accurate for the complete execution of the program/input pair.

4 Using the Simulation Points

After the SimPoint algorithm has chosen a set of simulation points and their respective weights, they can be used to accurately estimate the full execution of a program. The next step is to simulate in detail the interval for each simulation point, to collect the desired performance statistics.

4.1 Simulation Point Representation

SimPoint provides the simulation points in two forms:

Simulation Point Interval Number – The interval number for each simulation point is given. The interval numbers are relative to the start of execution, not to the previous simulation point. To get the start of a simulation point, subtract 1 from the interval number, and multiply by the interval size. For example, interval number 15 with an interval size of 10 million instruction means that the simulation point starts at instruction 140 million (i.e. (15-1)*10M) from the start of execution. Detailed simulation of this simulation point would occur from instruction 140 million until just before 150 million.

Start PC with Execution Count – SimPoint also provides for each simulation point the program counter for the first instruction of the interval and the number of times that instruction needs to be executed before starting simulation. For example, if the PC is 0x12000340 with an execution count of 1000, then detailed simulation starts the 1000th time that PC is seen during execution, and simulation occurs for the length of the profile interval.

4.2 Simulating the Points

After choosing the form of simulation points to use, each simulation point is then simulated. Two standard approaches for doing this are to use either fast-forwarding or checkpointing.

Fast-Forwarding – Sort the simulation points in chronological order. Fast-forward to the start of each simulation point. Simulate at the desired detail for the size of the interval. Repeat these steps, fast-forwarding from one point to the next combined with detailed simulation, until all simulation intervals have been collected.

Checkpointing – One advantage of SimPoint is that the state of a program can be checkpointed (e.g., using SimpleScalar's checkpoint facility) right before the start of each simulation point. This checkpointing allows parallel simulation of all of the simulation points at once.

4.3 Warmup

When using simulation points, an approach is needed for warming up the architecture state (e.g., the caches, TLBs, and branch predictor). The following are some standard approaches for dealing with warmup.

No Warmup – If a large enough interval size is used (e.g., 100 million instructions), no warmup may be necessary for many programs.

Assume Hit (Remove Cold Start Misses) – All of the large architecture structures (e.g., cache, branch predictors) make use of a warmup bit that indicates when is the first time an entry is used. If it is the first time, the access is assumed to be a hit or a correct prediction. One can also use a miss rate percentage (e.g., 10%) for these cold start misses, randomly assuming some percentage of the cold start accesses are misses. This a very simple method that provides fairly accurate warmup state, since the miss rates for these structures are usually fairly low [13, 6].

Stale State – This is a method of not resetting the architecture structures between simulation points, and instead they are used in the state they were in at the end of the prior simulation point we just fast-forwarded from [3].

Calculated Warmup – One can calculate the working set of the most recently accessed data, code and branch addresses before a simulation point. Then start the simulation of architectural components W instructions before the simulation point, where W is large enough to capture the working set size held by the architecture structures. After these W instructions are simulated, then reset any statistics and start the detailed simulation for that point. This approach brings the working set back into the architecture structures before starting the detailed simulation [2, 5].

Continuously Warm – This approach continuously keeps the state of certain architecture components warm (e.g., caches) even during fast-forwarding [14]. This is feasible if an infrastructure provides fast functional and structure simulation during fast-forwarding. Keeping the cache structures warm will increase the time it takes to perform fast-forwarding, but it is very accurate.

Either Calculated Warmup or Continuous Warmup provides the most accuracy, although we have found that for many programs Assume Hit and Stale State are fairly accurate.

4.4 Combining the Simulation Point Results

The final step in using SimPoint is to combine the weighted simulation points to arrive at an overall performance estimate for the program's execution. One cannot just use the harmonic mean for computing the overall miss rate, since we need to apply a weight to each sample.

Each weight represents the proportion of the total execution that belongs to its phase. The overall performance estimate is the weighted average of the set of simulation point estimates. For example, if we have 3 simulation points and their weights are [.22, .33, .45] and their CPIs are (CPI1, CPI2, CPI3), then the weighted average of these points is: $CPI = 0.22 * CPI1 + 0.33 * CPI2 + 0.45 * CPI3$

The weighted average CPI is the estimate of the CPI for the full execution.

4.5 Pitfalls to Watch for When Using Simulation Points

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

Calculating Weighted IPC – For IPC (instructions/cycle) we cannot just apply the weights as above. We first would need to convert all the simulated samples to CPI before computing the weighted average as above, and then convert the result back to IPC.

Calculating Weighted Miss Rates – To compute an overall miss rate, this must be calculated using the weighted average.

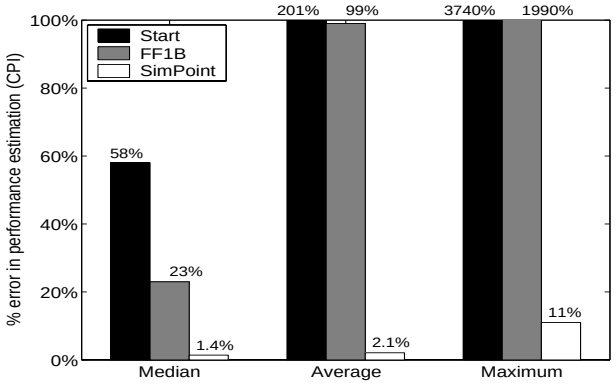

Figure 1: Simulation accuracy for the SPEC 2000 benchmark suite when performing detailed simulation using 1 million instruction intervals compared to the simulating the entire execution of the program. Results are shown for simulating from the start of the program's execution, for fast-forwarding 1 billion instructions before simulating, and for SimPoint. The first two methods simulate for 300 million instructions. The median results are for the complete SPEC 2000 benchmarks.

To compute the overall miss rate, we calculate the weighted average as is shown above for CPI, but replace the CPI estimates with the miss rate estimates.

Accurate Instruction Counts (No-ops) – It is important to count instructions exactly the same for the BBV profiles as for the detailed simulation, otherwise they will diverge. Note that the simulation points on the SimPoint website include no-ops in the instruction counts, so to reach a simulation point in a simulator, *all* executed instructions 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. This occurs when using the simulation point interval numbers. To avoid this, we suggest using the Start PC and Execution Count for each simulation point as described above.

4.6 Results

We now show the accuracy of using SimPoint for the complete SPEC 2000 benchmark suite and their reference inputs. Figure 1 shows the simulation accuracy results using SimPoint for the SPEC 2000 programs when compared to the complete execution of the programs. Results are shown for the median, average and maximum errors found. For these results, we set N (the number of instructions per interval) to 1 million, M (the maximum value of k) to 100, we try 7 different random seeds for each value of k, and we set X (the BIC score threshold, relative to the range of the observed BIC scores) to 90%.

For the non-SimPoint results, a simulation is run for 300 million instructions, which is over 3 times that of the SimPoint results provided. Figure 1 shows that starting simulation at the start of the program results in an average CPI error of 201% when compared to the full simulation of the program, whereas blindly fast forwarding for 1 billion instructions and then starting simulation results in an average CPI error of 99%. When

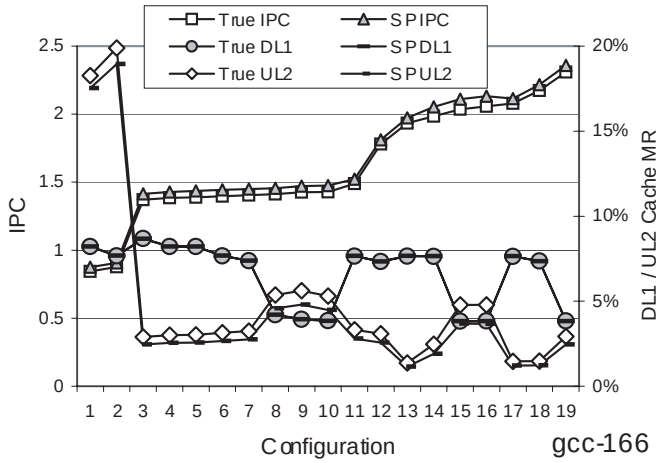

Figure 2: This plot shows the true and estimated IPC and cache miss rates for 19 different architecture configurations for the program `gcc`. The left y-axis is for the IPC and the right y-axis is for the cache miss rates for the L1 data cache and unified L2 cache. Results are shown for the complete execution of the configuration and when using SimPoint.

using the SimPoint algorithm to create multiple simulation points we saw an average CPI error of 2.1%. In comparison to random sampling approaches, we have found that SimPoint is able to achieve similar error rates requiring significantly (5 times) less simulation (fast-forwarding) time [10]. In addition, statistical sampling can be combined with SimPoint to create a phase clustering that has a low per-phase variance [10].

4.7 Architecture Independence

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 now examine how SimPoint tracks the relative change in hardware metrics across several different architecture configurations. 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 calculating the SimPoint estimates. We varied the configurations and the latencies of the L1 and L2 caches as described in [10].

Figure 2 shows the results across the 19 different architecture configurations we examined for three programs from the SPEC benchmark suite `gcc-166`. 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 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 2 shows that the 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. These results show that simulation points track the relative change in metrics between configurations. One interesting observation is that although the simulation results from SimPoint have a bias in the metrics, this bias is consistent and always in the same direction across the different configurations for a given program/input run. This is true for both IPC and cache miss rates. One reason for this bias is that SimPoint chooses the most representative interval from each phase, and intervals that represent phase changes may (if they occur enough) or may not (if they do not occur enough) be represented by a simulation point.

5 Creating Your Own Simulation Points

We now describe how to set the SimPoint parameters to create your own simulation points. The following are the parameters when using SimPoint:

Size of interval – The number of instructions per interval is the granularity of the algorithm. The interval size directly relates to the number of intervals, since the dynamic program length is the number of intervals times the interval size. Larger intervals allow more aggregate profile (basic block vector) representations of the program, while smaller intervals allow for more fine-grained representations. 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 the program.

Number of intervals – There should be a fair number of intervals for the clustering algorithm to choose from. A good rule of thumb is to make sure you are using 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, then decrease the interval size to obtain more intervals for clustering.

Max K – The maximum number of clusters (K from Section 2), along with the interval size, represents the maximum amount of simulation time that will be needed hen looking to choose simulation points. If SimPoint chooses a number of clusters that is close to the maximum allowed, then it is possible that Max K is too small. If this is the case and more simulation time is acceptable, it is better to double the Max K and re-run the SimPoint analysis.

Random Seeds – The k-means clustering algorithm starts from a randomized initialization, which requires a random seed. It is well-known that k-means can produce very different results depending on its initialization, so it is good to use many different random seeds for initializing different k-means clusterings, and then allow SimPoint to choose the best clustering. We have found that in practice, using 7 random seeds works well.

Number of iterations – The k-means algorithm iterates either until it hits a maximum number of iterations or until it reaches

a point where no further improvement is possible (whichever is less). In most cases 100 iterations is sufficient for the maximum number, but more may be required, especially if the number of intervals is very large compared to the number of clusters. A very rough rule of thumb is the number of iterations should be set to $\sqrt{N/k}$, where N is the number of intervals and k is the number of clusters.

Number of dimensions – SimPoint uses random linear projection to reduce the dimension of the clustered data, which dramatically reduces computational requirements while retaining the essential similarity information. SimPoint allows the user to define the number of dimensions to project down to. In our experiments we project down to 15 dimensions, as we have found that using it produces the same phases as using the full dimension. We believe this to be adequate for most SimPoint applications, but it is possible to test other values by looking at the consistency of the clusters produced when using different dimensions.

BIC percent – The BIC gives a measure of the goodness of the clustering of a set of data, and BIC scores can be compared for different clusterings of the same data. However, the BIC score is an approximation of a probability, and often increases as the number of clusters increase. This 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 (e.g. we use a BIC score of 90%). 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.

Creating simulation points with SimPoint is all about trading off accuracy for simulation time. If a user wants to limit the Max K to be small to limit simulation time, SimPoint can still provide accurate results. One reason for this is that for a given cluster a point close to the centroid (the average code usage of the cluster) is picked to represent the cluster. For example, limiting a clustering of `gcc` to be at most 300 clusters for an interval size of 1 million instructions, we have found the clusters formed to have very similar IPC. But, if the maximum number of clusters is only 10 using an interval size of 100 million, then some of the clusters will have varying IPC. The good news is that even in this case, since a point close to the centroid is chosen, this point represents the average code behavior of the interval, and this results in a low error rate.

We call the SimPoint algorithm we have focused on in this paper the Standard SimPoint approach. Another option for reducing simulation time is to use the Early SimPoint algorithm [10]. The Early SimPoint algorithm picks points which are close to the centroid, but not necessarily the closest, instead preferring points that are earlier in execution. This approach reduces the time it takes to complete the simulation of a program due to fast-forwarding.

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.

References

[1] M. Van 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*, March 2004.

[2] 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.

[3] 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.

[4] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, 2000.

[5] 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.

[6] 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.

[7] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[8] 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.

[9] 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, 2000.

[10] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[11] 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.

[12] 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.

[13] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1991.

[14] 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.