A Literature Survey of Effective Techniques to Reduce Simulation Time

Veynu Narasiman
narasima@ece.utexas.edu

Aater Suleman
suleman@ece.utexas.edu

1 Introduction

As microprocessor complexity increases, the time required to complete a thorough simulation of an entire benchmark will become a bigger and bigger challenge that computer architects have to face. This simulation time is orders of magnitude greater than the time required to run the benchmark on real hardware, thus, it is no longer feasible to simulate long benchmarks in their entirety. Given such a scenario, there is a definite need to reduce this simulation time so that computer architecture research can be performed efficiently in the future. In this paper, we present literature that suggests different techniques to reduce the amount of code being simulated, thereby greatly reducing the simulation time.

We are going to discuss six papers in the field. The first paper [14] is from a recent conference that identifies many of the various techniques that are being used to solve the aforementioned problem and also evaluates the effectiveness of each technique. This paper serves as an excellent introduction to the topic at hand. The conclusion of this paper is that sampling the benchmark trace is the most effective technique for fast simulation.

The remaining papers discuss two main types of sampling techniques. The first is statistical sampling, where randomly selected samples are used to represent the overall performance of a benchmark on a machine. The other method involves selecting only a single or a few strategically chosen (not random) samples, which are believed to be representative of the overall behavior of the program. This method, according to [14], is the best trade-off in terms of simulation time and accuracy.

The second paper [13] describes in more detail the statistical sampling approach, its methodology, and its effectiveness. The next three papers we present [12], [3], and [2] describe in detail the strategic approach to selecting samples. Although they use the same general approach, the difference between the techniques presented in these papers lies in the methods and criteria used to select the representative samples. We will describe and analyze them in detail later in the paper.

The last paper we present [10] may seem like a slight digression from the subject, but it is perhaps the one most importantly related to strategic sampling. This paper describes sampling at a different granularity (i.e. selecting benchmarks out of a benchmark suite that are most representative of the behavior of the entire suite). The paper lists many characteristics that can be attributed to a program. While some of these attributes have already been used by [12], [3], and [2] for strategic sampling, there is still room to use these metrics and their appropriate permutations to pick samples that are most representative for all sets of applications.

2 Summaries of Articles

2.1 Characterizing and Comparing Prevailing Simulation Techniques

This paper aims to describe and compare prominent simulation techniques that

are currently being used by computer architects to overcome the problem of long simulation times. According to the author, these techniques can be split into three major categories: 1)Reduced input set, 2) Truncated execution, 3) Sampling. The paper compares candidates from each of these categories based on three different characterization methods, namely Performance Bottleneck Characterization, Execution Profile Characterization, and Architectural Level Characterization. In addition to the three characterization methods, the paper also describes two other metrics that can be used as criteria to compare simulation techniques, namely speed versus accuracy and architectural dependence. The five comparison criteria as well as significant experimental results are discussed below.

2.1.1 Performance Bottleneck Characterization

Performance Bottleneck Characterization is performed using the Plackett and Burnam (PB) design. The PB design is used to calculate the effect of memory and processor parameters on the Instructions Per Cycle (IPC). These parameters are then ranked based on their PB magnitudes and vectorized. A single vector is computed for each candidate. These vectors are then used to compare each candidate with the reference input. The difference between each candidate and the reference input set is quantized using the Euclidean distance.

In this experiment, it was observed that the sampling techniques (SMARTS and SimPoint) were the alternatives with the least Euclidean distance from the reference input, hence they were the most representative of the reference input. The Euclidean distance between reduced input set and reference input varied significantly across all benchmarks. The paper concludes, after further analysis, that reduced input set in fact simulates a very different program as compared to the reference input set and therefore is not representative of the reference input. The truncated execution candidates also performed poorly because the portion of the program that was executed was chosen arbitrarily, thus, there were no guarantees regarding the representativeness.

2.1.2 Execution Profile Characterization

Execution Profile Characterization is performed by recording the Basic Block Execution Frequencies (BBEF) of a program. A basic block is a section of code that is executed from start to finish with one entry and one exit. To compute a BBEF, whenever a basic block is executed, the basic block counter is incremented by the number of instructions that were executed in that block. Once the BBEF vectors for the reference input and each candidate have been computed, they are compared using the χ^2 statistic.

The observed results were comparable to the results obtained using Performance Bottleneck Characterization. The reduced input and truncated execution techniques had very different execution profiles when compared to the reference input. However, sampling techniques had an execution profile similar to the reference input set.

2.1.3 Architectural Level Characterization

Architectural Level Characterization is performed by recording a set of architectural metrics such as IPC, branch prediction accuracy, and cache hit rate. These metrics are vectorized and the vectors are then compared with the vector obtained from the reference input set using the Euclidean distance. The results obtained were the same as the results of Performance Bottleneck and Execution Profile Characterization.

2.1.4 Speed vs. Accuracy

There is always a trade-off between simulation time and accuracy which makes this metric very important. This metric is computed by calculating the Manhattan distance between the CPIs of each candidate and the reference input set, then plotting it against

the normalized simulation time (normalized with respect to the simulation time of the reference input). It was concluded that sampling techniques performed significantly better than both reduced input set and truncated execution.

2.1.5 Architectural Configuration Dependence

The last metric the paper uses for comparison is Architectural Configuration Dependence. The accuracy of an ideal technique should remain constant over all architectural configurations. Experiments were performed in order to gauge the configuration dependence of the candidates. The results obtained suggest that this metric is correlated with the accuracy computed in the three characterization methods described earlier.

Based on the five criteria, sampling the benchmarks proved to be the most effective technique to reduce the length of a benchmark for detailed execution. We will build upon this conclusion, and the remaining papers discussed in this review will further explore the prevailing sampling techniques.

2.2 SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling

This paper describes methods to statistically sample full length benchmarks. It describes a framework, SMARTS, that accelerates the simulation of a benchmark by selectively measuring in detail only an appropriate subset of a benchmark. The SMARTS approach involves applying statistical sampling theory in order to select an appropriate number of program subsets and ensure representativeness of the entire program.

This paper asserts that systematic sampling is preferred to random sampling for sampling benchmarks because benchmarks do not show conceivable patterns, and also because systematic sampling is easier to implement in event-driven simulators. SMARTS takes a periodic approach where CPI is sampled at fixed intervals. Only the instructions in the sampling unit are executed in detailed simulation mode, whereas others are simulated using a functional simulator. SMARTS can estimate the average CPI of the entire benchmark by averaging the CPIs of all the samples. More importantly, SMARTS can also provide a measure of representativeness that can be computed by calculating the confidence of the CPI estimate. In summary, SMARTS prescribes an exact procedure to generate an accurate performance estimate by measuring only a minimal subset of instructions.

Architects face several challenges when using SMARTS. The most important is the *cold start problem* which expresses concern about the inaccurate microarchitectural state of the machine (due to incomplete simulation) prior to the execution of each sampling unit. This problem is overcome by performing functional simulation on all the instructions between the sampling units, and detailed simulation on a few instructions before each sampling unit. Only the microarchitectural state of the machine is updated during the functional simulation phase. The architectural state is updated only in detailed simulation mode. There are certain discrepancies associated with this warm-up approach. The state thus computed may not be accurate as it does not include the effects of out-of-order execution and speculative event ordering. According to the author, these effects are minimal and can be ignored.

The SMARTS framework takes three inputs as parameters, U (the sampling unit size in instruction), W (detailed warming unit size in instructions), and N (the number of sampling units in benchmark). These three parameters can be varied in order to adjust the speed versus accuracy trade-off. The authors suggest that a sampling unit of 1000 instructions is most optimum for any benchmark. They also define two variables n and k where n is the number of samples collected from the benchmark and k is the number

of sampling intervals between two consecutive samples. Therefore, n=N/k. The authors provide a statistical method to determine n or k for a certain required accuracy. However, the parameter W is difficult to estimate. In order to keep the errors introduced due to bad warm-up below 1.5%, some SPEC 2000 benchmarks require a W greater than 50,000. Although 50,000 is a small percentage of the benchmark itself, it can still add considerable overhead to the simulation time. The authors did further analysis to limit W. It is stated that functional warming reduces the value of W signifcantly and puts an uppderbound on W. This upper bound is equal to the product of store-buffer depth, memory latency in cycles, and the maximum IPC. This bound is an upper-bound that is applicable only to the worst case scenarios, hence a relatively smaller W (2000 or 4000) can be used.

The authors also evaluate the performance and accuracy of SMARTS on the SPEC 2000 benchmark. SMARTSim (a simulator developed by the authors) is only 50% slower than an average functional only simulator. This shows that the results are promising in terms of both performance and accuracy. To conclude, the authors compare their results to another prevailing technique, SimPoint, that uses strategic sampling (SimPoint is described in much greater detail in the summary of [12] below). The authors observed that SMARTS outperforms SimPoint in accuracy. In addition, they point out an important fact that SimPoint does not estimate the representativeness of the samples like SMARTS does, making SMARTS a more attractive alternative.

Now we will move to our discussion of SimPoint by summarizing a recent paper published on SimPoint in ASPLOS X.

2.3 Automatically Characterizing Large Scale Program Behavior

This paper introduces a new technique of analyzing programs to help understand the large scale program behavior of even the most complex applications. The clustering tool described in this paper is called SimPoint and is available at [1]. Using Basic Block Vectors combined with clustering, SimPoint accurately identifies similar intervals within a large application and strategically chooses sample intervals that are highly representative of the entire program. Only thoroughly simulating (detailed simulation) these sample intervals instead of the entire program greatly reduces the simulation time but still yields accurate results.

2.3.1 Basic Block Vectors

At the heart of the analysis lies the concept of a Basic Block Vector, an idea originally presented in an earlier work [11] by the same authors. A Basic Block Vector is a single dimensional array with one entry corresponding to each basic block within a program (a basic block can be defined as a section of code that is executed from start to finish with only one entry and one exit). Each entry within the vector contains the number of times the corresponding basic block was executed during a certain interval, times the number of instructions within that basic block. The vector is then normalized by dividing each entry by the sum of all the entries. This paper used an interval size of 100 million instructions, thus for each interval of 100 million instructions within a large application, a new Basic Block Vector is tabulated. These vectors can now be compared in order to determine how similar different intervals are to each other.

Basic Block Vectors are compared by calculating either the Euclidean or Manhattan Distance between them. The Euclidean Distance is found by taking the sum of the square of the difference between corresponding entries, while the Manhattan Distance represents the sum of the absolute value of the difference between corresponding entries. A similarity matrix can now be computed where an entry (x,y) in the matrix represents the similarity (Manhattan

Distance) between the xth and yth intervals. By analyzing the similarity matrix, one can identify similar intervals and identify phases within a program. This paper calculates the similarity matrix for some common applications (gzip, bzip, and gcc) and shows that the phases identified using the similarity matrix line up quite closely with the actual program behavior.

2.3.2 Clustering

Now that we know the similarity among different intervals, we can group the intervals into clusters. This paper uses a partitioning algorithm known as K-means [6] in order to cluster the intervals. K-means clustering works as follows:

1) Initially, K intervals from the set of all intervals are randomly selected to be the cluster centers.
2) Next, each interval is assigned to the cluster center that it is closest to (Manhattan distance is the smallest).
3) Lastly, each cluster center is recalculated as the average of all the members of that cluster (the members were assigned in step 2).
4) Steps 2 and 3 are repeated until cluster membership stops changing between successive iterations.

In the above algorithm, K represents the number of clusters. This paper runs the above algorithm for K values 1 to 10, and then uses the Bayesian Information Criterion [8] to select the K value that is the best fit for the data. By the end of the algorithm, all of the intervals are a member of 1 of the K clusters, and each cluster has a cluster center (also called centroid) which is the average of all the intervals in that cluster.

Multiple simulation points can now be chosen based upon this information. One interval from each cluster is chosen to represent the entire cluster. The interval that is chosen is the one that is closest to the cluster center. In addition, each interval chosen is accompanied by a weight based on the size of the cluster it is representing. Together, the selected intervals and their corresponding weights accurately represent the entire program.

The results obtained using this technique were very promising. The IPC calculated from simulating only the chosen intervals and taking into account their weights, was within 3% of the actual IPC obtained by simulating the program in its entirety, justifying the effectiveness of this technique.

Several other SimPoint-based techniques have also been introduced by researchers for strategic sampling. The primary difference between such proposals and the one presented in this paper is the use of other metrics instead of the Basic Block Vector. Next, we will discuss some of these related techniques.

2.4 Structures for Phase Classification

This paper explores the effectiveness of different program structures' ability to capture phase behavior. The SimPoint clustering tool is used, but instead of sending the tool Basic Block Vector data, different criteria are used and their accuracy is evaluated.

2.4.1 Control Flow Structures

Instead of breaking up a program into basic blocks and using Basic Block Vectors, this paper introduces the idea of using loop vectors, and procedure vectors. For loop vectors, the entire program is divided into loops, and there is an entry in the vector corresponding to each loop in the program. The number of times each loop is entered during a certain interval is recorded, and that vector is than weighted and normalized just as explained earlier for Basic Block Vectors. Procedure vectors work the same way except for the fact that the program is divided into static procedures, and there is an entry in the vector corresponding to each procedure. To evaluate the accuracy of using different program structures for phase classification, this paper calculates the covariance

of CPI for each cluster SimPoint reported, and then computes an average covariance of all the clusters. This average covariance is one number that represents how accurate the phase classification really is. In addition, the CPI obtained from simulating only the intervals selected by SimPoint is compared to the actual CPI obtained when simulating the program in its entirety.

The results showed that the average covariance computed using loop vectors and procedure vectors was comparable to that of Basic Block Vectors. In addition, the percent CPI error rate was only slightly greater for procedures and loops when compared to basic blocks. However, the loop vectors and procedure vectors are much smaller than Basic Block Vectors, and therefore remain a valid alternative to using basic blocks.

2.4.2 Profiling Instruction Mix

In this section, memory instruction vectors, and opcode vectors were introduced as other options to profile a program. Memory instruction vectors keep track of the number of times every load or store is executed during a certain interval. Opcode vectors have 64 entries (since there are 64 opcodes in the Alpha ISA) each representing the number of times that particular opcode was executed during a certain interval.

Once again, the results showed that the average covariance using memory instruction and opcode vectors was comparable to that of Basic Block Vectors, but the CPI error rate was in general greater.

2.4.3 Profiling Registers

This section introduces a method of tracking the uses/definitions of a given register during a certain interval of execution. The register use vector contains 32 entries (since the Alpha ISA has 32 registers) each containing the number of times that register was used during a certain interval. Likewise, the register definition vector tracks the number of times a certain register was defined during a given interval. Just as before, these vectors are sent to SimPoint for clustering and phase identification.

The results showed that tracking register definitions produced results comparable to using basic blocks with respect to average covariance, as well as CPI percent error.

2.4.4 Stride Profiling

This section proposes the idea of capturing the distribution of strides between loads/stores to memory. For example, for local stride profiling, if a certain load/store is currently accessing memory location 8, but previously that same load/store instruction accessed location 4, we increment the 4th (8 minus 4 = 4) entry in the local stride vector. This vector is, as usual, populated over a given interval. A global stride vector is also created in a similar fashion, but instead of tracking the stride of individual loads/stores, the stride between adjacent memory accesses is recorded. In addition, a variation of these vectors is introduced by hashing the index into the local/global stride vector with the lower bits from the PC. These are simply known as local/global vectors with PC hash. Once again, these vectors (one for each interval) are sent to SimPoint for phase classification.

The results showed that the average covariance was lowest for local stride with PC hash (global covariance was 7% higher), and were comparable to that of Basic Block Vectors. However, the CPI percent error was once again, greater than that obtained by using basic blocks.

2.4.5 Key Findings

Ultimately, this paper showed that using register definition vectors and also loop vectors are both attractive alternatives to using basic blocks. Both vectors are considerably smaller than Basic Block Vectors and thus are easier to manage and require fewer resources to track. Both also yield results comparable to those of basic blocks.

Now we move to the next paper which describes another program metric that can be used to detect the phase behavior of a program. The paper summarized below describes the use of memory access behavior as a pattern detection mechanism.

2.5 Choosing Representative Slices of Program Execution for Microarchitectural Simulation

This paper introduces a new method of selecting representative slices of a program to be used for simulations. The method described in this paper is tailored to producing accurate results for cache miss rate. In this paper, a program is broken up into slices each of which are 10 million instructions long. These slices are then grouped together to form classes. A representative slice from each class is selected and used for detailed simulation.

2.5.1 Metrics Used

To classify the slices into groups, the individual slices must be profiled according to some criteria. This paper introduces the idea of Data Reuse Distance (RDI) which is a measure of the number of instructions executed in between two accesses to the same memory location. This value is calculated for each memory access in the program. This gives us information about the temporal locality of the slice. Calculating the RDI values for many different cache line sizes gives us information about the spatial locality of the slice. For a given line size, and for each integer n, the number of memory accesses for which the RDI value is between 2^n and $2^{n-1} - 1$ is recorded. This is done for each slice of the program. This information is then used to group the slices into classes.

2.5.2 Classification Method Used

This paper used a hierarchical grouping algorithm to form classes. This particular study used a tool known as CHAVL which implements the Likelihood Linkage Analysis (LLA) classification method [4] [5]. The data that needs to be grouped into classes is the RDI data tabulated as described in the previous section. After classification, a representative slice from each class is chosen. The chosen slice is the one whose Euclidean distance to the center of the class is least. This Euclidean distance is known as the "representativeness" of the selected slice.

This paper also defines an indicator, known as the *wmdc* (weighted mean distances from centers) which is the weighted average (according to number of members in the class) of the "representativeness" of each class. This number serves as an indication of how representative the selected slices are of the entire program. The smaller the *wmdc*, the more representative the selected slices are of the entire program.

2.5.3 Results

The method for choosing representative slices presented in this paper was compared to the method of choosing one big slice of execution (truncated execution) and also to the systematic statistical sampling method. The relative percent error was calculated by using the cache miss rate obtained by simulating only the selected slices and the actual cache miss rate obtained by simulating the entire program. This study showed that in general, the percent relative errors were less using the method presented in this paper, than when using the big slice or the systematic statistical sampling method. This led the authors to two important conclusions:

1) The metrics chosen in this paper are well suited for cache simulations.
2) This approach is really able to select a few slices that accurately represent the entire program.

2.6 Measuring Program Similarity

This paper aims to present a methodology for measuring similarity between programs using architecture independent criteria. Past

approaches to measuring program similarity had primarily relied on architecture dependant metrics, and thus did not produce consistent results over a wide range of architectural configurations. The end goal of this study is to identify similarities between groups of programs such that only one good representative of the group can be simulated instead of the whole group. If a subset of a benchmark suite is chosen at random, there is risk that the subset may solely consist of benchmarks with similar characteristics. This can skew the results of computer architecture studies. To avoid the aforementioned problem, the authors propose a systematic experimental and statistical approach to better characterize a group of programs.

The authors first define 29 different metrics on which programs were to be evaluated. These metrics can be divided into categories such as instruction mix, branch taken or not taken data, basic block size, temporal locality, spatial locality, and dependency information. The authors used these 29 microarchitecture independent metrics to compare programs. The procedure they use to select a subset from a whole suite of benchmarks is described below.

Each metric is evaluated for every program in the benchmark suite. Once the metrics have been computed, the data is vectorized. These 29-dimentional vectors are difficult to comprehend, so a technique called Principal Component Analysis is used to reduce them down to 6 dimensions. These 6-dimensional vectors are then used as input to K-means algorithm for grouping. One drawback of the K-means algorithms is that K (the number of clusters) has to be specified beforehand. This problem is overcome by exploring various values of K in order to find the optimum value K. The optimum grouping is selected on the basis of Bayesian Information Criteria (BIC). Once completed, this methodology yields a subsetting of the set of programs where each member of a group has similar characteristics. A representative set of programs can then be picked by selecting one program from each group. The program that is picked from each group is the one closest to the center of the group.

The above mentioned procedure is further illustrated by an example of subsetting the SPEC CPU2000 benchmark suite. The microarchitecture independent characteristics were measured for each benchmark. Details of these results can be seen in [9]. The SimPoint program is then used to obtain two different types of groupings for the benchmark suite. One is based on all 29 characteristics and the other is based on 7 data locality characteristics only. Using all 29 characteristics, the optimum K value (number of clusters) computed was 8. Using only the 7 data locality characteristics, the number of clusters was determined to be 9.

The evaluation methodology used by the authors is similar to the methodology described in [14] as the Architectural Characterization. First, the average IPC of the entire suite is calculated and then compared to the average IPC calculated using the chosen subset. The two IPCs differ by a very small amount on two different machine configurations, validating both the accuracy and microarchitecture independence of this technique. Second, a similar comparison is done on the basis of L1 D-cache miss-rate and the results again turned out to be promising. Another set of results presented is the sensitivity of accuracy to the number of representatives chosen. The results are intuitive, as we increase the number of representative elements, the average cache miss rate of the subset approaches the average cache miss rate of the whole suite.

This paper also provides an analysis of all the SPEC benchmarks suites. The results show that all the SPEC CPU benchmarks overlap in terms of instruction locality characteristics, implying that these characteristics have not changed over the past decade. Similar results are drawn for the branch characteristics and instruction level parallelism. It can also be seen that the floating point benchmarks have longer dependency chains

compared to integer benchmarks and floating point branches are mostly loop branches compared to branches in integer programs. Data locality characteristics however exhibit a slightly different trend. It can be seen that data locality has been getting worse over time (with the exception of a few outliers). The paper has also run a similarity analysis across all the SPEC benchmark suites, and the optimum number of clusters was determined to be 12. Lastly, the paper finds that CPU 2000 is the most diverse among the all the SPEC suites that have surfaced.

3 Evaluation

In this literature survey, we analyzed some of the recent papers that describe techniques for reduction of simulation time. We chose papers that advocated the use of statistical or profile driven simulation to reduce the simulation time. All these papers, except [14], described and evaluated a different methodology to pick the samples. The ideas, overlapping in some instances, provide a good overview of benchmark sampling. The ideas described in these papers were innovative and can provide a good lead for anyone planning to start research in this area. Each of the papers had their individual strengths and weaknesses which we will discuss in the paragraphs to follow.

The paper "Characterizing and Comparing Prevailing Simulation Techniques" does an excellent job of choosing characterization methodologies. The methodologies cover a wide range of program behaviors. In addition, each methodology is described in great detail making it easier for the reader to understand the experimental techniques. However, the paper does not provide the details of the experimental setup adequately. For example, the paper speaks of 43 processor and memory parameters, but does not provide a list of these parameters that were used in PB design. Such a list could have added more value to the results presented. The paper actually refers to [15] for a description of these parameters, but even there, no such listing can be found.

The results provided by the author are intuitive to the reader and similar to those provided in other related literature. For instance, the reduced input technique evaluated using the Processor Bottleneck Characterization rendered the same conclusion stated in [7]. This ensures validity of results. Overall this paper was of value to literature in this subject and served as an ideal overview for this survey.

The paper from Wunderlich *et al.* on SMARTS was a strong paper and it provided not only experimental results but also provided analytical explanations for these results. The reason that makes SMARTS a powerful technique is its ability to estimate the representativeness of the selected samples. In addition, SMARTS provides the ability to adjust the accuracy versus speed-up trade-off before the simulation is run, by altering the sampling unit size and sampling frequency. This flexibility is not offered by their competitor SimPoint. Results from [14] state that SMARTS outperformed SimPoint in terms of accuracy, but had a greater simulation time. This was due to the fact that SMARTS requires detailed execution of more instructions.

The main drawback of SMARTS is the *cold start problem* that arises due to short sample unit size and leads to the need for warm-up. Another disadvantage, also related to warm-up, is that all simulators running SMARTS need to support both functional and detailed simulation. This special need limits the scope of this technique. However, SMARTS is a very powerful technique which, according to [14], should be the choice of any architect who is most concerned about accuracy rather than simulation speed.

The paper from Sherwood *et al.* titled "Automatically Characterizing Large Scale Program Behavior" provided a description of SimPoint. The paper does an excellent job of explaining the concepts as well as the

methodology at work in the SimPoint tool. Even though some of the mathematical concepts in the paper were hard to convey, the explanations were still clear. Some of the figures in the paper were hard to comprehend, even in the presence of descriptions. Overall, SimPoint is a powerful idea and the use of basic block vectors with SimPoint is also clever. SimPoint performs well in achieving accurate numbers for IPC for most of the SPEC benchmarks but it does not perform well for memory intensive benchmarks like mcf, leaving room for improvement in the area of strategic sampling.

"Structures for Phase Classification," also from Sherwood *et al.*, furthers the ideas from the SimPoint paper by trying other alternatives in place of Basic Block Vectors. The text was clear and easy to understand. The paper has an important value as it increased the number of options for researchers to investigate. An important conclusion was that register usage can be tracked instead of basic blocks to achieve a similar level of accuracy. Since there are a limited number of registers, this data is easier to collect and manage. Some of these other metrics presented in this paper may also help overcome the deficiency of SimPoint in estimating performance for memory intensive workloads.

The paper presented by Lafage and Seznec provides an alternative to SimPoint for strategic sampling. This paper uses data stream characteristics instead of instruction stream to perform phase classification. It uses a hierarchical clustering algorithm and uses data related metrics, which makes this paper different from other papers in this survey. The idea is to use temporal and spatial locality as a metric for profiling benchmarks. The authors did not do a satisfactory job at describing the method they used to compute these complex metrics. There is even a mathematical error in §4.1.3 of the paper which creates further confusion. The paper could have been a more significant contribution if the concepts had been explained better. The paper also provides an indicator of representativeness which SimPoint did not, making this paper a novel contribution. The ideas in this paper can be applied to future research in generating more representative samples.

The paper from Phansalkar *et al.* is an important contribution to the subject. The most significant contribution of this paper, which can lead to further research ideas in the field of strategic sampling, is the identification of 29 microarchitecture independent metrics. Although the idea is not presented in this paper, a subset of these metrics can be used instead of Basic Block Vectors as input to a tool like SimPoint. The paper also provides reference to [9] where they not only provide the list of these 29 metrics but also a table where the values for these metrics have all been computed for programs from SPEC 2000 and former SPEC suites. Overall, this paper is easy to read and comprehend, and provides some important insights concerning the SPEC benchmark suites.

4 Conclusion

This literature survey discusses papers related to the sampling of benchmarks. We provide a summary of six papers underlining the key concepts and contributions of each paper. We also provided our evaluation of these papers. The following important conclusions can be drawn from this survey and used in research related to this field.

1) Sampling is the most practical technique to reduce simulation time, yet still maintain accuracy
2) Statistical sampling, although more accurate, has more serious issues related to execution time and warm-up
3) Strategic sampling can lead to promising results, but the results can be more or less representative based on the metrics used to select the samples
4) There is still room from improvement in the area of statistical and strategic sampling

References

[1] Simpoint. http://www.cs.ucsd.edu/calder/simpoint/, March 2005.

[2] Thierry Lafage and Andre; Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. pages 145–163, 2001.

[3] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for phase classification. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 2004.

[4] I. C. Lerman. Foundations of the likelihood linkage analysis (iia) classification method. *Applied Stochastic Models and Data Analysis*, pages 7:63–76, 1991.

[5] I. C. Lerman. Likelihood linkage analysis (lia) classification method: An example treated by hand. *Biochimie*, 1993.

[6] J. MacQueen. Some methods for classification and analysis of multivariate data. in 5th berkeley symposium. volume 1, pages 281–297, 1967.

[7] A. J. Klein Osowski and David J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, 2002.

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

[9] A. Phalsanker, A. Joshi, L. Eeckhout, and L. John. Measuring program similarity. Technical Report TR-050127-01, The University of Texas at Austin, 2005.

[10] A. Phansalkar, A. Joshi, L. Eeckhout, and John L. Measuring program similarity: Experiements with spec cpu benchmark suites.

[11] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14. IEEE Computer Society, 2001.

[12] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57, 2002.

[13] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.

[14] Joshua J. Yi, Sreekumar V. Kodakara, Resit Sendag, David J. Lilja, and Douglas M. Hawkins. Characterizing and comparing prevailing simulation techniques. Laboratory for Advanced Research in Computing Technology and Compilers, February 2005.

[15] Joshua J. Yi and David J. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *ICCD*, pages 462–, 2002.