Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 05-044

PASS: Program Structure Aware Stratified Sampling for Statistically Selecting Instruction Traces and Simulation Points

Sreekumar V. Kodakara, Jinpyo Kim, Wei-chung Hsu, David J. Lilja, and Pen-chung Yew

December 30, 2005

PASS: Program Structure Aware Stratified Sampling for Statistically Selecting Instruction Traces and Simulation Points

ABSTRACT

As modeled microarchitectures become more complex and the size of benchmark program keeps increasing, simulating a complete program with various input sets is practically infeasible within a given time and computation resource budget. A common approach is to simulate only a subset of representative parts of the program selected from the complete program execution. SimPoint [1,2] and SMARTS [10] have shown that accurate performance estimates can be achieved with a relatively small number of instructions.

This paper proposes a novel method called Program structure Aware Stratified Sampling (PASS) for further reducing microarchitecture simulation time without losing accuracy and coverage. PASS has four major phases, consisting of building Extended Calling Context Tree (ECCT), dynamic code region analysis, program behavior profiling, and stratified sampling. ECCT is constructed to represent program calling context and repetitive behavior via dynamic instrumentation. Dynamic code region analysis identifies code regions with similar program phase behaviors. Program behavior profiling stores statistical information of program behaviors such as number of executed instructions, branch mispredictions, and cache miss associated with each code region. Based on the variability of each phase, we adaptively sample instances of instruction streams through stratified sampling.

We applied PASS on 12 SPEC CPU2000 benchmark and input combinations and achieved average 1.46 % IPC error bound from measurements of native execution on Itanium-2 machine with much smaller sampled instruction streams.

Keywords: Simulation, Stratified Sampling

Author List:

Name	Email	Phone
Sreekumar V. Kodakara*	sreek@ece.umn.edu	763-218-5024
Jinpyo Kim+	jinpyo@cs.umn.edu	612-626-9742
Wei-Chung Hsu+	hsu@cs.umn.edu	612-625-2013
David J. Lilja*	lilja@ece.umn.edu	612-625-5007
Pen-Chung Yew+	yew@cs.umn.edu	612-625-7387

+Department of Computer Science and Engineering
University of Minnesota,
200 Union St,
Minneapolis MN - 55414

*Department of Electrical and Computer Engineering
University of Minnesota,
200 Union St,
Minneapolis MN - 55414

1 INTRODUCTION

Cycle accurate simulation has been essential in investigating a set of design parameters of new microarchitectures and in evaluating the performance impact of respective compiler optimizations. As the complexity of modeled microarchitecture and the size of benchmark programs keep increasing, it is practically infeasible to simulate the complete program execution over multiple configurations of microarchitecture within a given time and computation resource budget. Therefore, we must simulate only representative parts of a program by judiciously selecting [1,2] or randomly sampling instruction streams from the complete program execution [9,10].

Furthermore, as new microarchitectures enable more aggressive static and dynamic compiler optimizations [26], cycle accurate simulation will also be used for evaluating the interaction between compiler optimization and new microarchitectures. In previous work on sampling based accelerated simulation [1,2,9,10], simulation samples are selected from equally divided intervals or arbitrary location in the full dynamic instruction streams regardless of program structure boundaries such as loops and procedure calls. Selecting simulation samples aligned with program structure can be more effective for evaluating specific compiler optimizations. This is because a new compiler optimization may change the number of instructions generated. Therefore, previously selected SimPoints [1,2] may no longer be applicable.

Random sampling is a pure statistical approach for solving the problem and is known to give accurate estimates of the behavior of the benchmark program [10,22]. EXPERT [17] showed that repetitive program behavior can be exploited to accelerate simulation and still obtain accurate estimates. In this paper, we exploit the program calling context and repetitive behavior with dynamic code region analysis and behavior profiling to obtain highly accurate estimates with smaller sample size when compared to Random Sampling.

This paper proposes a novel sampling method called Program structure Aware Stratified Sampling (PASS) for significantly reducing microarchitecture simulation time without losing accuracy and coverage. PASS involves four major phases consisting of building Extended Calling Context Tree (ECCT), dynamic code region analysis, program behavior profiling, and stratified sampling. ECCT is constructed to represent program calling context and loops via dynamic instrumentation. Dynamic code region analysis identifies runtime constructed code regions

showing similar phase behaviors. Program behavior profiling stores statistical information such as number of executed instructions, branch mispredictions, and cache misses associated with each code region. Finally, based on the variability of each dynamic code region, we adaptively select instances of instruction streams though stratified sampling.

ECCT is an extension of the Calling Context Tree (CCT) proposed by Ammons et. al.[23]. Loop nodes are added to reflect program repetitive behavior in addition to calling context. Calling context and loops are also exploited for reducing CPU power consumption in [24,25]. W. Liu and M. Huang [17] also explored program structures to accelerate simulation using static profiling. Our ECCT can accommodate various profiling data such as IPC, cache misses, branch predictions with the calling context and repetitive behaviors. It allows us to more effectively select instruction streams according to specific performance characteristics. For example, if we are interested in the impact on instruction caches, selecting instructions from a stable phase may be inappropriate since instruction cache misses often occur at phase transition time.

The primary contributions of our paper are as follows:

- We show that highly accurate performance estimate can be achieved with small number of sampled instruction stream from full execution through Program structure Aware Stratified Sampling.
- We show that stratified sampling can be effectively used for accelerating microarchitecture simulation if stratum is properly selected through dynamic code region analysis over ECCT.
- We show that ECCT can effectively capture dynamic calling context and repetitive behaviors and is appropriate to accommodate program behavior profile such as number of retired instructions, branch mispredictions, and cache misses for feedback.

The rest of this paper is organized as follows. Section 2 describes the details of PASS framework to identify representative code regions and sample instruction streams effectively. Section 3 describes evaluation methodology used in our proposed new sampling technique. Section 4 presents and discusses the results and compares them to other techniques. Section 5 describes related work and the motivation of our work. Finally, Section 6 summarizes the work.

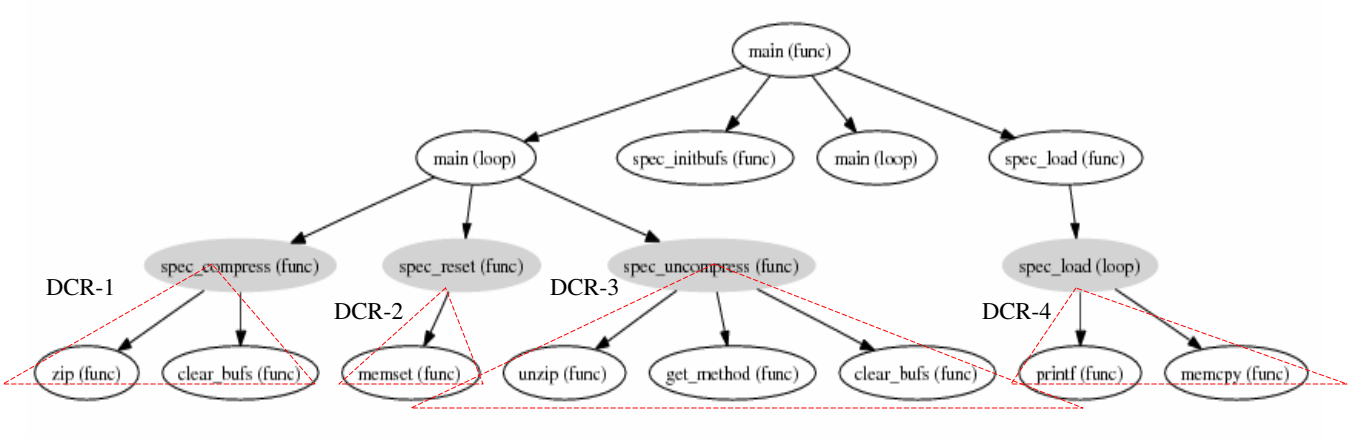
2 PASS FRAMEWORK

In this section, we describe the PASS framework consisting of construction of ECCT, dynamic code region analysis, program behavior profiling, and stratified sampling.

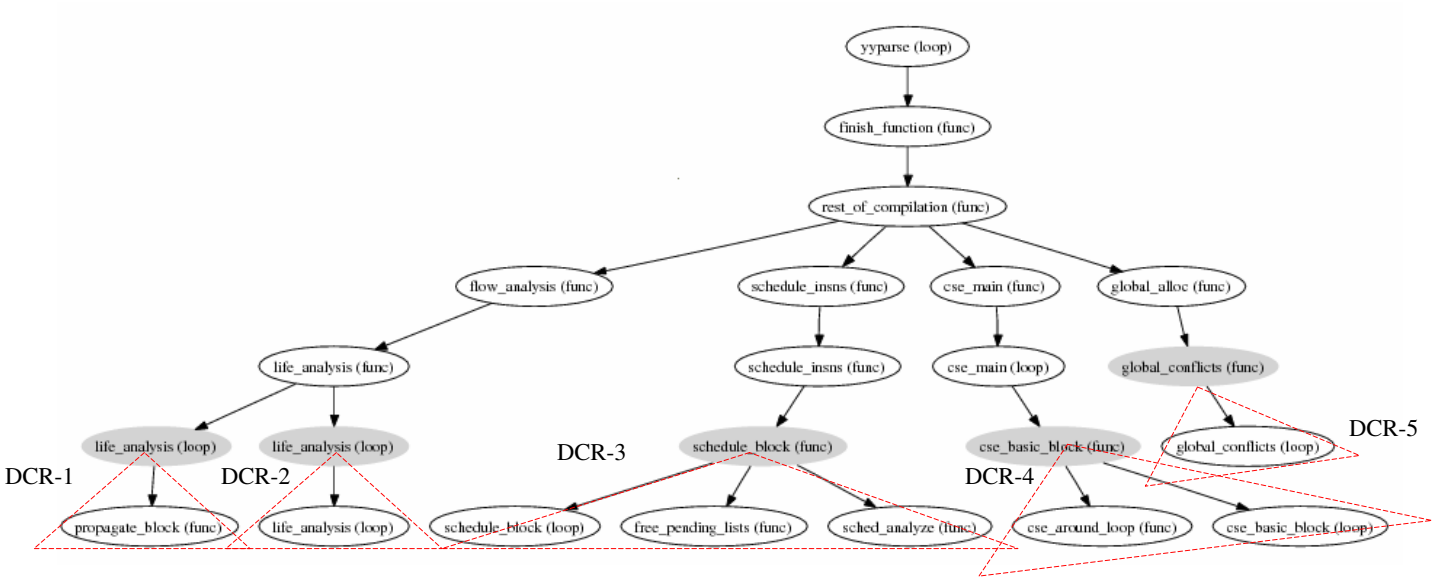
2.1 Extended Calling Context Tree

Calling Context Tree (CCT) was first proposed by Ammon et al [23]. CCT is a directed graph G=(N,E), where N is the number of nodes that represent procedures in the program and E are the edges that connect the procedures. For example, if a procedure *proc1* is called from another procedure *proc2*, the graph will include two nodes *proc1* and *proc2* with a directed edge connecting *proc2* to *proc1*. In CCT, only those procedures that are called during the execution of the program are present. Nodes representing procedures in CCT are *context sensitive* [16]. If a procedure *proc1* is called from procedures *proc2* and *proc3*, the graph will contain two different nodes for *proc1*. Creating unique nodes for procedures for each context makes the graph a tree. We add nodes representing loops to CCT, and call it Extended Calling Context Tree (ECCT). All properties of the nodes representing procedures in CCT are also applicable to loop nodes in ECCT.

Figure 1(a) and 1(b) show part of the ECCT obtained for gzip and gcc respectively. The nodes that have (func) and (loop) prefix are procedure and loop nodes respectively. Each node in ECCT is annotated with statistical information about the execution of the node and all the nodes in the subtree with the node as the root. Depending on the application, information about number of instructions executed, cache misses, branch miss predictions, and so on, could be recorded in each node of the tree. In our experiments, the cumulative number of dynamic instruction executed in the node and all the nodes under it is recorded. For example in Figure 1(a), the node *spec_compress*, will have the total number of instructions retired in *zip* and *clear_bufs* in addition to the instructions retired in *spec_compress*. ECCT is generated by instrumenting the benchmark. Details of the instrumentation are presented in section 4.



(a) gzip



(b) gcc

Figure 1. Partial ECCT of gzip and gcc is shown. The grey nodes are the root of the sub-tree that forms the DCR and the triangles mark the DCR's.

2.2 Dynamic Code Region

A Dynamic Code Region (DCR) is a set of nodes and their subtrees in ECCT. function In Figure 1(b), for example, *schedule_block (func)* and its child nodes *schedule_block (loop)*, *free_pending_lists (func)* and *schedule_analyze (func)* can be grouped together and considered as one dynamic code region. Depending on the target application, we can identify DCR's that have desirable characteristics for the application. In this paper the target application is accelerated microarchitecture simulation using stratified sampling. In the next section we will

give a brief review of the stratified sampling technique and define the desirable characteristics of the DCR. In section 3.4 we describe the analysis algorithm that can be used to identify a set of DCR's from the ECCT.

2.3 Stratified Sampling

Sampling consists of selecting some parts of a population to observe so that one may estimate something about the whole population [19]. A sampling unit is an element of the population whose parameter can be measured. In microarchitecture simulation, a sequence of instructions can be considered as a sampling unit. The measurable parameter of the sampling unit are the performance metrics like IPC, branch mis-prediction rate, and cache miss rate that we are interested to estimate for the entire program execution. There are many classifications of sampling techniques depending on the way sampling units are selected for measurement. The two techniques that we discuss in this paper are Simple Random Sampling and Stratified Random Sampling. In Simple Random Sampling, the sampling units are selected at random from the population. In Stratified Random Sampling, the population is first partitioned into regions or strata and random sampling units are selected independently from each stratum. The total sample size can be split among different strata using Neyman allocation (also called optimal allocation) or proportional allocation [19,20]. In Neyman allocation, the number of samples that are drawn from each stratum is directly proportional to the variation of the performance metric within the stratum, while in proportional allocation, the number of samples for each stratum is allocated based on its size. Neyman allocation would give better results than proportional allocation if the variance in the parameter is higher in some strata when compared to others. Figure 2 shows the formulas used for Stratified Random Sampling and Simple Random Sampling. Formulas 1 and 4 are used to find the estimate of the population total from the measured samples in Stratified Random Sampling and Simple Random Sampling respectively. L is the total number of strata and N_h is the total number of instructions executed in each strata. n is the number of samples measured in Simple Random Sampling and N is the number of samples present in the population. Formulas 2 and 5 are used to find the sample size that should be measured from the population such that the error in the final estimate is within r * 100% of the actual value with (1-α) * 100% confidence. σ_h is the sample standard deviation of the performance metric within strata h and γ is the coefficient of variation (standard deviation divided by mean).

In order to get significant gains in using Stratified Random Sampling for simulation, the dynamic instruction stream of a benchmark should be partitioned such that within stratum variance of the parameter is very small and the means of different strata are different from one another. In PASS, we split the dynamic instruction streams of the benchmark program based on DCR. The idea behind splitting the dynamic instruction stream based on DCR is that a group of functions and loops that call from one another can show similar behavior across the execution of the program.

Stratified Random Sampling

(1) Estimate of the Population Total $= \tau_{str} = \sum_{h=1}^{L} x'_h$

where $x'_h = N_h \bar{x}_h$

(2) Total Sample Size $= n_{str} \approx \dfrac{\left(\frac{z^2_{1-(\frac{\alpha}{2})}}{N^2}\right)\left(\sum_{h=1}^{L} \frac{N_h^2 \sigma_h^2}{\pi_h \bar{X}^2}\right)}{r^2 + \left(\frac{z^2_{1-(\frac{\alpha}{2})}}{N^2}\right)\left(\sum_{h=1}^{L} \frac{N_h^2 \sigma_h^2}{\bar{X}^2}\right)}$

where $\pi_h = \dfrac{n_h}{n}$

(3) Stratum Size (Neyman Allocation) $= n_h = \dfrac{n N_h \sigma_h}{\sum_{h=1}^{L} N_h \sigma_h}$

Simple Random Sampling

(4) Estimate of the Population Total $= \tau_r = \left(\dfrac{N}{n}\right) \sum_{i=1}^{n} x_i$

(5) Total Sample Size $= n_r = \dfrac{1}{\frac{r^2}{z^2 \gamma^2} + \frac{1}{N}}$

Figure 2. Formulas for Stratified Random Sampling and Simple Random Sampling

In the next section we describe an algorithm that automatically identifies a set of DCR's from the ECCT.

2.4 Dynamic Code Region Analysis

In this section, we describe an algorithm that automatically identifies non-overlapping DCR's with high code coverage and relatively stable behavior, which are desirable for accelerated microarchitecture simulation.

The input to the algorithm is an ECCT of the program that has cumulative number of retired instructions stored in each node. The algorithm also requires *Coverage* and *Granularity* of the DCR as input parameters. In ECCT, the

instructions that correspond to statements other than a procedure call or a loop in the parent node (if-then-else conditions, statements etc) are not included in the cumulative instruction count of the child node. Thus the sum of the cumulative instruction counts of all child nodes will be less than the cumulative instruction count stored in the parent node. We define coverage to be the ratio of the sum of the instruction counts in each DCR to the instruction count in the root node of ECCT. We define the total number of instructions retired in a DCR to be the Granularity of the DCR. Granularity of DCR's was introduced to prune away very small DCR's from the final set.

The search algorithm is iterative. It starts from the root of the tree and proceeds in a breadth first order. The frontier of the search is initialized with the root node. During any iteration of the search, one node is selected from the frontier of the search. The selected node is removed from the frontier and its child nodes that satisfy minimum size constraint is added to the frontier. Next, the ratio of the sum of the cumulative number of instructions of all nodes in the frontier and the cumulative instruction count present in the root node is calculated. If this ratio is greater than or equal to the specified coverage value, the new frontier is accepted. If the ratio is less than the specified coverage value or if no child node has the specified minimum granularity, the parent node is marked and is not considered for further exploration. This completes an iteration of the search algorithm. When no unmarked nodes are present in the frontier, the algorithm terminates. The nodes in the frontier at the termination of the algorithm are root nodes for the sub-tree that forms the final set of DCR. To avoid the search from going deeper and deeper in one region of the tree, the node in the frontier of the search that is closer to the root node is given higher priority. . Figure 1 (a) and 1 (b), show the DCR's obtained for gzip and gcc respectively. The root node of each DCR is marked grey. The dotted lines under the grey node mark the nodes that are included in each DCR.

The complexity of the search algorithm is linear in the number of nodes in the tree. The search algorithm is given in Figure 3.

```
1. Initialize the current frontier to NULL.
2. Add root node to the current frontier
3. Select an unmarked node that is closest to the root node from the
current frontier. If no such nodes are present, stop.
4. Iterate through the child nodes of the selected node
5. If at least one child node has a granularity that is greater than the
specified minimum granularity, proceed to step 6, else mark the selected
node and go to step 3.
6. Create a new frontier with all nodes apart from the selected node added
to it. Add the child nodes of the selected node to the new frontier.
7. Find the sum of the cumulative number of instructions in the new
frontier and divide it by the sum present in the root node.
8. If the ratio is less than the minimum coverage value, mark the selected
node. Delete the new frontier and go to step 3.
9. If the ratio is greater than or equal to the minimum coverage value,
delete the current frontier. Make the new frontier as the current frontier
and go to step 3.
```

Figure 3. The DCR Analysis Algorithm

3 EVALUATION METHODOLOGY

We describe the tools used and the experiment set ups for evaluating the effectiveness of the PASS method in this section. The results of IPC estimate using PASS will be discussed in section 5.

3.1 Tools Used for Evaluation

Pin and Pfmon [13, 15] were used in our experiments for evaluating PASS and comparing it with Simpoint and Random sampling. Pin is a dynamic instrumentation framework developed at Intel for Itanium and x86 processors [13]. We used Pin to dynamically instrument binaries so that we may build ECCT with function and loop nodes. Pfmon is used to read the performance counters of the Itanium processor [15]. Pfmon was also modified to measure the performance for simulation points related to different DCR's in the program. Due to real machine measurements, there could be noise in the collected performance data. We identified two kinds of noise sources in our measurements - the noise introduced by routines in pfmon during measurement and random noise introduced by the scheduler in the OS. The noise from the interrupt service routine of pfmon device driver is relatively small. The noise was measured over multiple runs and the average noise was subtracted from the final data. The random noise was minimized by repeating the data collection 7 times and using the median value. All the data reported in

this paper were collected from a 900 Mhz Itanium-2 processor with 1.5 M L3 cache running Redhat Linux Operating System version 2.4.18-e37.

3.2 Benchmarks

Ten benchmarks from the SPEC CPU2000 benchmark suite (7 integer and 3 floating point benchmarks) were evaluated. Those benchmarks were selected partly because they are known to exhibit some phase behavior. Reference input sets were used for all benchmarks. Two integer benchmarks, gzip and perl, were evaluated with 2 different input sets. A total of 12 benchmarks and input set combinations were evaluated. All benchmarks were compiled using gcc (version 3.4) at O3 optimization level.

3.3 Evaluation of PASS

Using PASS for performance analysis is a four step process – ECCT construction, dynamic code region analysis, behavior profiling and stratified sampling. The following sections describe these steps in detail.

3.3.1 Profiling and analysis

The first step is to construct the ECCT of the benchmark program. ECCT is constructed by identifying function calls and loops during the execution of benchmark program. Function calls are identified by *call* and *ret* instructions. Many modern architectures have call/ret instructions defined in the ISA. Detecting recursive function calls is a special case to be considered when building the ECCT. Recursive function calls are represented as one node in ECCT and the complete recursion tree is considered as one DCR during analysis. Loops are identified using backward branches. An instruction counter is maintained in each node of the ECCT and it records the cumulative number of retired instructions in the node and the sub-tree under it. Each node in the ECCT is assigned a unique node ID. The name of the function is obtained from the debug information stored in the binary. The generated ECCT is dumped to a file at the end of the instrumentation.

The second step is the analysis of ECCT. In our implementation of the analysis routine, two parameters of requirements, the minimum granularity of each DCR and the desired coverage of the DCR's, are taken as inputs. The algorithm performs a search on the ECCT and identifies a set of DCR's that satisfies the granularity and coverage requirements. In our experiments, we set the granularity to be 10 Million instructions and the coverage

to be 0.95. Figure 4 shows the number of DCR's obtained for these parameter settings for different benchmark programs. As expected, programs with larger code base like gcc and perl have higher number of DCR's than programs like gzip and mcf. The node ID's of the final set of DCR are got from the analysis routines and will be used as inputs to our measurements.

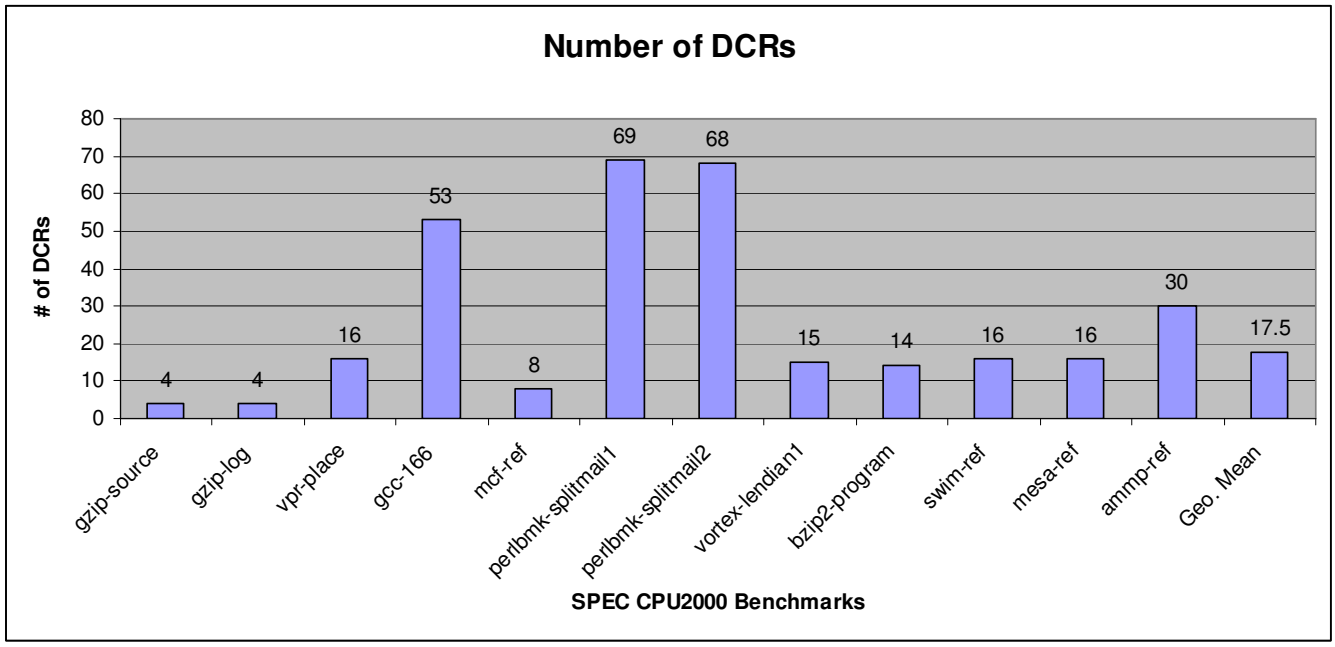


Figure 4. Number of DCRs obtained using the analysis algorithm for all benchmark programs. The granularity was set to 10 Million instructions and the coverage is set to 0.95.

The third step is to collect the performance profile of each DCR in the benchmark program. The performance profile is collected in order to obtain an estimate of the variation of the performance metric within each DCR. This would enable us to effectively allocate sample size to different strata. Random samples of equal length are chosen from the dynamic instruction streams of each DCR and the performance for each sample is measured. In this paper each sample is 200,000 instructions long. A total of 200 samples were measured from each benchmark to obtain its performance profile. These 200 samples were distributed among different strata using proportional allocation. For example, if the benchmark had two DCR's of size 10 Million and 30 Million instructions, the number of samples allocated for the two DCR's is 50 and 150 samples respectively. Two performance metrics

namely IPC and Branch mispredictions ratio were measured for the selected samples. The standard deviation of the performance metric is calculated for each DCR, which is used in the measurement step to allocate samples.

Figure 5 shows the CoV value of the number of cycles for the top 5 DCR's for all benchmark programs. CoV is the ratio of standard deviation and mean. Higher CoV value implies higher variability in performance within the DCR. The DCR-1 to DCR-5 is ordered in descending order of the size of DCR. It can be seen that DCR's with larger sizes need not necessarily have higher variability. Thus sample size allocation based on Neyman allocation would give better results when compared to proportional allocation [19,20]. The profiling step is computationally expensive due to instrumentation, but we would like to emphasize that it is a onetime effort and the cost could be amortized over multiple simulation runs.

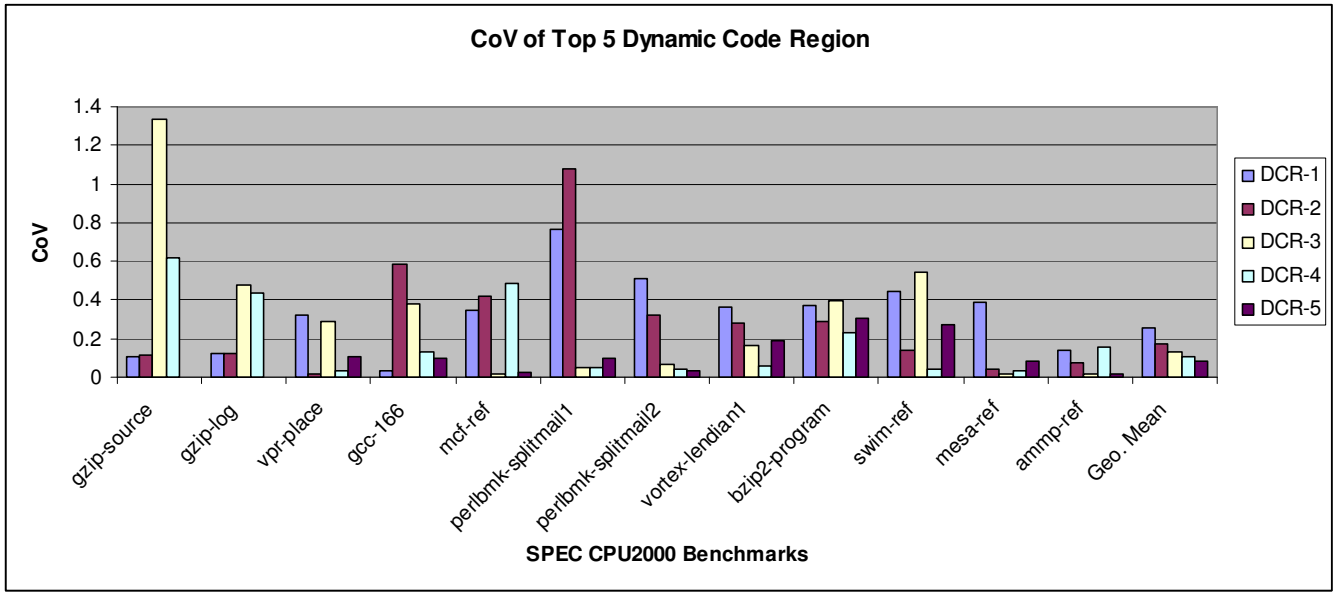


Figure 5. CoV of Top 5 DCRs of all benchmark programs obtained in the profiling step.

3.3.2 Measurement with stratified sampling

The fourth step is to measure the performance of the benchmark program and to calculate an estimate of the performance metric using stratified sampling theory. Measurement is an iterative process. For the first iteration, the total number of samples measured for each benchmark program is a fixed number. This fixed number of samples is distributed among different DCR's using Neyman allocation. Random samples of equal length are chosen from each DCR and the performance metric is measured. The measurements are then used to find the total

number of samples required to estimate the performance of the benchmark for a given confidence and error probability. If the new sample size is less than the current sample size, then there is no need to do further measurement. The estimation of each DCR is then weighted and combined together to get the final performance estimate of the program. If the new sample size is greater than the current sample size, the program has to be measured again with the new sample size. Typically, a maximum of two feedback iterations are sufficient to obtain the final result.

3.4 Warm-up and Measurement Issues

3.4.1 Warm-up

We used real machine measurements to evaluate PASS. Hence we did not need to deal with warm-up issues typically required in microarchitecture simulation. There have been many proposals in the literature to handle warm-up efficiently [7,8,9] . One of those could be used in simulating samples selected by our method to warm-up the caches and other microarchitecture state. Finding warm-up points for selected samples based on DCR analysis is also possible, but is beyond the scope of this paper.

3.4.2 Real Machine Measurement

Since real machine is used for measurements, the node ID's cannot be directly used to sample the DCR. The node ID's needs to be converted to a set of points marked by dynamic instruction counts, that corresponds to the dynamic instruction streams of different DCRs. We re-ran the pin instrumentation routines to dump the start and stop instruction counts that correspond to random samples from different strata. These instruction counts were then used to measure the performance of different DCRs in a benchmark. It should be noted that, the second instrumentation is an artifact of using real machine for doing performance measurement. If we had used a simulator instead, we could have constructed the ECCT during the simulation and switch to detailed performance simulation when the DCR needs to be simulated is identified.

3.4.3 Granularity of Measurement and over sampling

In order to reduce the random noise in our measurements, the length of each sample was set to 200,000 instructions. This restriction had an implication in measuring small DCR's in some benchmarks. There were

DCR's in some benchmarks that had significant contribution to the overall program execution, but the length of each instance of the DCR would be less than 200,000 instructions. In those cases, a set of instances of the DCR were grouped together such that the total dynamic instruction count was greater than 200,000 instructions. This group was measured and the measured value was considered as the measurement of one instance of the DCR. In effect, we over sampled those small DCR's for the sake of accuracy in our measurements.

4 RESULTS

In this section we present the results from using PASS and compared against the Simple Random Sampling and the SimPoint techniques. We implemented Simple Random Sampling and SimPoint in our framework and measured the performance on Itanium-2 machines. In the original SMARTS paper, the sampling units from the benchmark were selected systematically, while in this paper we select it randomly. Apart from this difference, the formulas used for calculating the total sample size and performance estimate are exactly the same to that used in the SMARTS paper. Due to the computational complexity of clustering used in SimPoint we restricted the size of each SimPoint to 10 Million instructions. The length of each sampling unit for PASS and Simple Random Sampling is set to 200,000 instructions.

Two data points are shown for PASS. The first result - PASS-No Feedback – are the measurements obtained in the profiling step. Two hundred samples were measured from each benchmark to obtain the results in PASS-No feedback. The 200 samples were distributed among the strata using proportional allocation. For smaller DCR's, the sample size allocated in proportional sampling could be less than 1. In order to get a reliable estimate of the performance for such DCR's, 5 samples were measured from them. This is the reason why the total sample size in Table 1 for PASS-No feedback is greater than 200. The second result - PASS-Feedback - is obtained from the measurement step in PASS. In PASS-Feedback, variance in IPC for each stratum is obtained from the profiling step and formula's shown in Figure 1 is used to find the total sample size and the sample size of each stratum. Random-Feedback refers to measurements taken for Simple Random Sampling technique.

4.1 Error in estimating IPC

Figure 5 and 6 shows the error ratios in the estimate of IPC and Branch mis-predictions in PASS, Simple Random Sampling and SimPoint when compared to the full run of the benchmark programs. It can be seen that PASS-No Feedback, which uses a fixed total sample size and proportional allocation, is generally pretty accurate. In vpr,ammp and swim the error in PASS-No Feedback is higher than 5%. We can attribute that to bias in the performance data of some strata due to dynamic events like cache misses, branch mis-predictions etc. In those cases, according to stratified sampling theory, increasing the total sample size and/or allocating the number of samples based on the variation in the performance of each stratum will reduce the error to the specified error bounds. PASS-Feedback uses the performance profile stored in each DCR to reallocate number of samples and to adjust the total sample size. With feedback mechanism, Random-Feedback and PASS-Feedback meet 5% error bound while SimPoint shows over 5% error in vpr-route, perlbmk-splitmail2, and bzip2-program. It should be noted that statistical bounds on the results guarantees its reliability.

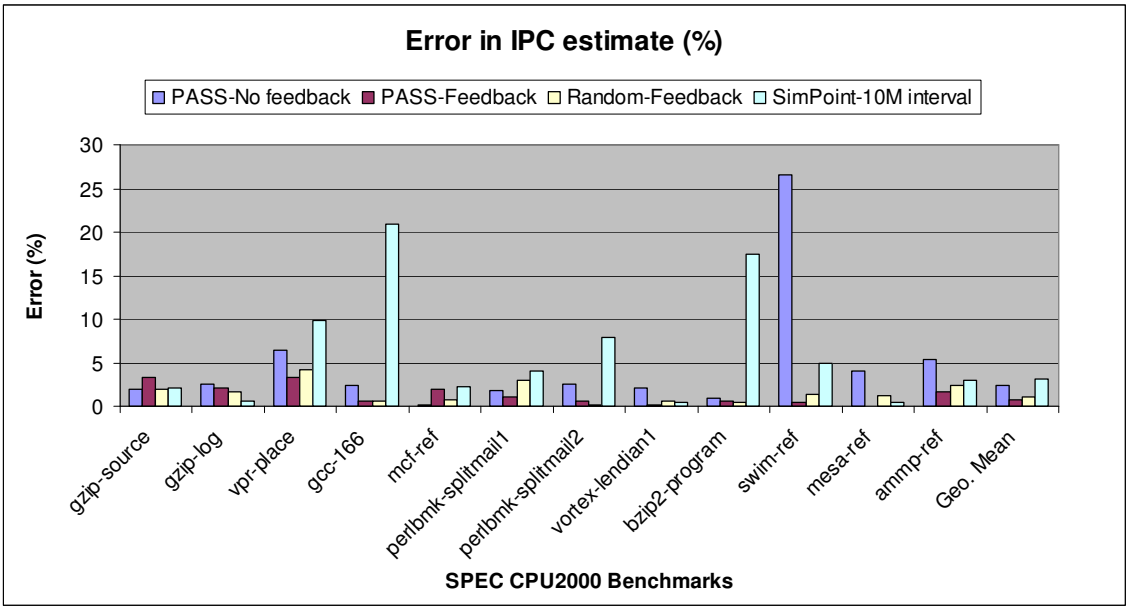


Figure 5. Comparison between the error in IPC estimate between PASS, Simple Random Sampling and SimPoint.

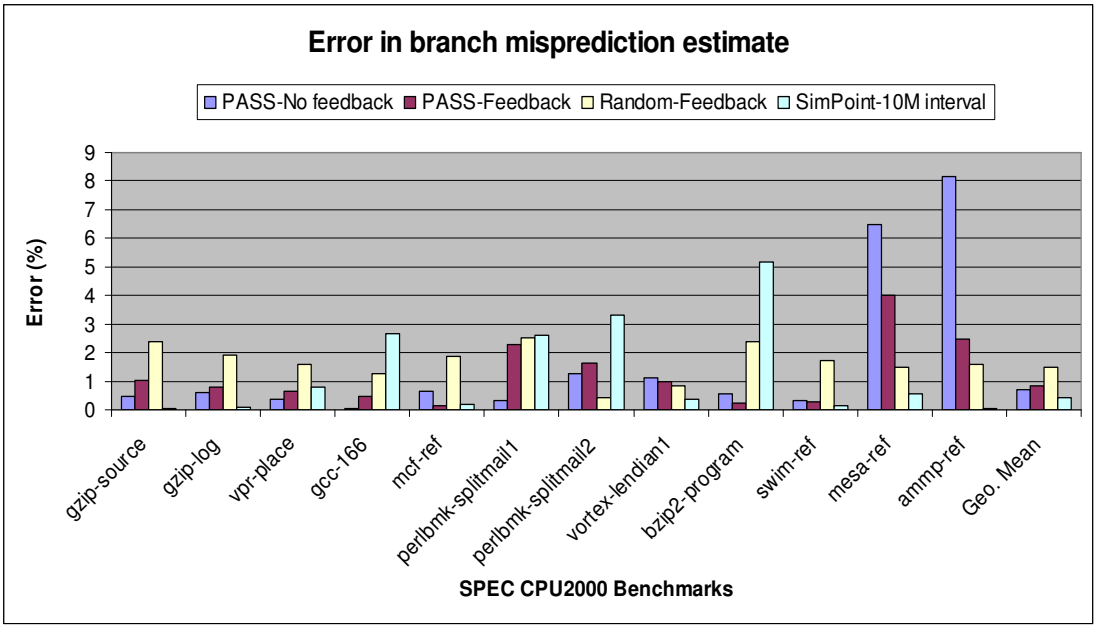


Figure 6. Comparison between the error in branch mispredictions estimate between PASS, Simple Random Sampling and SimPoint.

In the next section we show that the number of instructions that need to be measured for PASS to achieve the error bound is significantly less when compared to Simple Random Sampling.

4.2 Sampling points and total number of instructions needed for measurements

Table 1 compares the number of instructions measured in each simulation technique to obtain the IPC results shown in Figure 5. Each sample is 200,000 instructions long in PASS and Random sampling. In SimPoint, sample size is 10 Million instructions. The column, Relative ratio to PASS-Feedback in Table 1, is obtained by dividing the number of instructions measured in Random-Feedback or SimPoint with that measured in PASS-Feedback. On an average, total sample size measured in PASS-Feedback is 3 times smaller when compared to the sample size in Random-Feedback. For example, in swim, the sample size in PASS-Feedback is 8 times smaller than that in Random-Feedback. Swim is a floating point benchmark and is loop intensive. The some loops are known to have higher cache misses. The analysis algorithm in PASS identified 4 main loops in swim and partitioned them into strata. Only 128 samples, allocated using Neyman allocation, was sufficient to estimate the performance of of

swim. This shows that DCR based grouping of the dynamic instruction sequence is an effective technique in reducing the overall variability in the benchmark program.

Table 1. Number of Sampling Points measured for different simulation techniques.

Benchmarks	PASS – No Feedback	PASS-Feedback	Random-Feedback		SimPoint	
			# of Sample Points	Relative ratio to PASS-Feedback	# of Sample Points	Relative ratio to PASS-Feedback
Gzip-source	209	29	325	11.21	9	15.52
Gzip-log	208	104	200	1.92	10	4.33
vpr-place	270	208	200	0.96	6	2.16
gcc-166	388	290	439	1.51	9	1.55
Mcf-ref	227	197	225	1.14	9	2.28
Perlbmk-splitmail1	477	809	1100	1.36	8	0.56
Perlbmk-splitmail2	462	245	324	1.32	8	1.84
vortex-lendian1	242	211	200	0.95	8	2.13
Bzip2-program	227	209	426	2.04	10	2.15
swim-ref	257	128	1125	8.79	9	3.52
mesa-ref	260	212	200	0.94	6	2.12
Ammp-ref	326	74	312	4.22	9	6.08
average	296.08	226.33	423.00	3.03	8.42	3.69

Figure 7 shows the fraction of the benchmark program that is required to be measured for all techniques. In PASS-Feedback, on an average, only 0.02% of the dynamic instruction stream needs to be measured, when compared to 0.04% for simple random sampling and 0.05% for Simpoint.

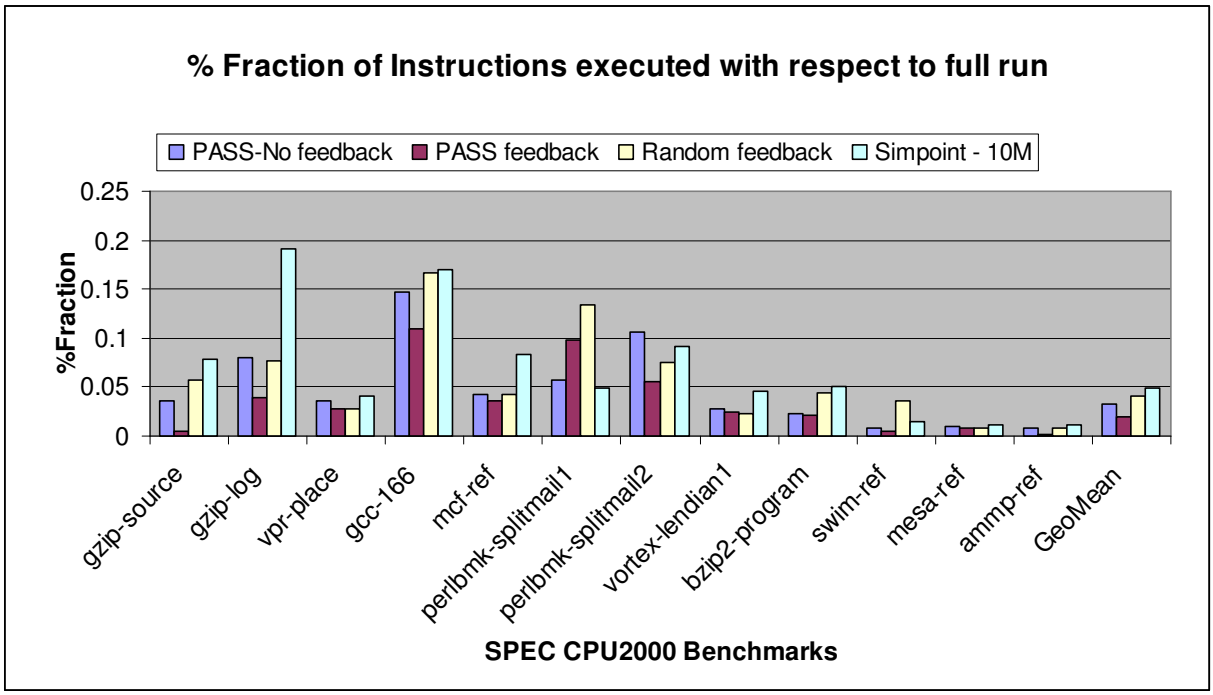


Figure 7. Fraction of the total number of instructions executed in PASS, Simple Random Sampling and Simpoint when compared to full run

5 RELATED WORK

In this section, we discuss related work on sampling techniques for accelerated simulation and identifying representative parts of a program.

Sampling techniques for accelerated simulation Accelerating microarchitecture simulation with statistical sampling was studied by T. Conte et al [12,21]. Sampling may be categorized into two types: *probability sampling* and *non-probability sampling*. Probability sampling requires that samples are selected from a randomized process. SMARTS [10] belongs to this category. In non-probability sampling, samples are chosen subjectively after determining targets for sampling. Usually, targets for sampling in programs are representative instruction streams that are repeatedly executed across full execution.

SMARTS [10] uses simple periodic sampling with statistical confidence. It is a simple and easy to implement and it doesn't require any prior information regarding program behavior. It also provides statistical confidence in the results. However, it can not take advantage of repetitive behavior in programs. Profiling time to determine the optimal sample size is very time consuming and the different sample size may be needed according to new

configuration parameters of microarchitecture. R. Wunderlich et. al [11] studied of applying stratified sampling to further reduce simulation time and number of simulation points and showed that optimal stratified sampling could reduce required measurement requirement by 43x over simple random sampling. They examined BBV stratification and optimal stratification and concluded that we need better stratification approaches to provide a clear advantage over simple random sampling.

SimPoint [1,2] finds representative portions of the programs using Basic Block Vectors (BBV) gathered through full instrumentation. It clusters the BBV using the K-Means algorithm and finds representative BBV from each cluster. It simulates the selected portion of the program and weighs the result by the relative coverage of each BBV cluster. This method is attractive because it achieves very accurate estimate with a small number of samples. However, selecting only one sample from a representative portion potentially could lead to large errors in the estimation for some programs. Since the time to cluster BBVs increases very quickly as the dimension of BBV and the number of BBVs increase, it could be very time consuming to explore various parameters of clustering to decide right sample size for every benchmark and combinations of input sets. Choosing the right sample size could be even more difficult because the sample size and the number of samples show different correlation with simulation errors in each benchmark program.

EXPERT [17] identifies representative portions of the programs by marking loops and function calls in static code section through profiling. It uses reduced input sets to collect profiles and applies statistical sampling to those selected portions of the program.

Identifying representative parts of a program Understanding dynamic program behavior has been a key to identify representative parts of a program and develop new microarchitecture and compiler optimization. E. Duesterwald et. al. [4] studied metrics derived from hardware counters to characterize and predict program behavior and its variability for incorporating prediction into adaptive systems. T. Sherwood et. al [2] proposed a novel method to automatically characterize large scale program behavior by using BBV based clustering. This technique has been further studied to show correlation between program code signature and actual behavior through simulation [14] and measurement from real machines [3, 6]. In [18], we showed that online tracking of

the change in code signature can be used to effectively identify and predict phase changes. We used a hardware stack to track the signature of code regions and a lookup table to store unique code signatures. The stable stack bottom in the hardware stack is compared against the lookup table every one million instructions to identify a phase change. By sampling the stack every one million instructions, the hardware is looking for phase changes that is *local* to the region of execution and at a granularity of one million instructions. This combined with limited size of the table can result in a larger number of phases. In this paper, we build an ECCT by instrumenting the benchmark program to track every calling context and loops during full execution of the program. The ECCT is then analyzed *offline* using the proposed search algorithm to find dynamic code regions. The search algorithm in this paper has a *global* view of the execution of the program. Hence, unlike the online hardware, it can identify the minimum number of dynamic code region to stratify program execution.

6 CONCLUSION

Statistical sampling has received considerable attention from researchers for accelerating microarchitecture simulation. Stratified sampling has potential to reduce simulation time over simple random sampling due to the presence of phase behavior in programs. A lack of efficient stratification method made it difficult to exploit this potential. In this paper, we proposes a novel sampling method called Program structure Aware Stratified Sampling (PASS) that exploits calling context and loops to stratify the execution of the program. We applied PASS methodology for sampling instruction streams on 12 SPEC CPU2000 benchmark and input combinations. We achieved average 1.46 % IPC error bound from measurements of native execution on Itanium-2 machine. To achieve this accuracy, the sample size selected by PASS are about 3 times smaller than samples selected by random sampling, and about 3.7 times smaller than samples selected by SimPoint at 10 Million instructions per sample.

This technique also can be used for effectively sampling instruction traces for detailed performance bottleneck analysis and trace driven simulation. Since our technique follows on stratified sampling theory, the analyzed result can provide statistical confidence with consistent error bound which is important for reliability of the result.

7 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, September 2001.

[2] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In Architectural Support for Programming Language and Operating Systems – X, October 2002.

[3] 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 MICRO-37, December 2004.

[4] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In International Conference on Parallel Architectures and Compilation Techniques, October, 2003

[5] E. Schnarr and J.R. Larus. Fast out-of-order processor simulation using memorization. In Architectural Support for Programming Languages and Operating Systems – VIII, October 1998.

[6] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies. The Fuzzy Correlation between Code and Performance Predictability. In the 37th International Symposium on Microarchitecture, December 2004.

[7] J. Haskins and K. Skadron. Accelerated warmup for sampled microarchitecture simulation. In ACM Transactions on Architecture and Code Optimization (TACO), 2(1), March 2005.

[8] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. Technical Report UCSD-CS2004-0803, University of California, San Diego, November 2004.

[9] T.F. Wenisch, R.E. Wunderlich, B. Falsi, and J.C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Technical Report CALCM 2004-3, Carnegie Mellon University, November 2004.

[10] T. Wenisch, R. Wunderlich, B. Falsafi and J. Hoe, SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling, In Proceedings of the 31th Annual International symposium on Computer Architecture, IEEE Computer Society, 2003.

[11] R.E. Wunderlich, T.F. Wenisch, B. Falsafi and J.C. Hoe. An Evaluation of Stratified Sampling of Microarchitecture Simulations. In Proceedings of the 3rd Workshop on Duplicating, Deconstructing, and Debunking, June 2004

[12] T.M Conte, M.A. Hirsch, K.N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In IEEE International Conference on Computer Design: VLSI in Computers and Processors, October 1996.

[13] PIN – A Dynamic Binary Instrumentation Tool. http://rogue.colorado.edu/Pin.

[14] J. Lau, S. Schoenmackers, and B. Calder. Structures for Phase Classification. In IEEE International Symposium on Performance Analysis of Systems and Software, March 2004.

[15] http://www.hpl.hp.com/research/linux/perfmon.

[16] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufman, 1997.

[17] W. Liu and M. Huang, EXPERT: expedited simulation exploiting program behavior repetition. In Proceedings of the 18th annual international conference on Supercomputing, June 2004.

[18] J. Kim, S.V. Kodakara, W.-C. Hsu, D.J. Lilja and P.-C. Yew. Dynamic Code Region (DCR)-based Program Phase Tracking and Prediction for Dynamic Optimizations. In International Conference on High Performance Embedded Architectures and Compilers, November 2005. *To be appeared.*

[19] S.K. Thompson, Sampling, 2nd Ed, Wiley Series in Probability and Statistics, 2002.

[20] W.G. Cochran, Sampling Techniques, 3rd Ed, Wiley, 1977.

[21] T. M. Conte, M. A. Hirsch and W. W. Hwu, Combining Trace Sampling With Single Pass Methods for Efficient Cache Simulation, In IEEE Transactions on Computers, vol. C-47, no. 6, Jun. 1998.

[22] J.J. Yi, S.V. Kodakara, R. Sendag, D.J. Lilja, and D.M. Hawkins. Characterizing and comparing prevailing simulation techniques. In HPCA-11, February 2005.

[23] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In Proceedings of the ACM SIGPLAN`97 Conference on Programming Language Design and Implementation (PLDI), June 1997.

[24] G. Magklis, M. L. Scott, G. Semeraro, D. A. Albonesi, and S. Dropsho, Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In Proceedings of the International Symposium on Computer Architecture, June 2003.

[25] C.-H Hsu and U. Kermer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2003.

[26] S. Adve, et al. Changing Interaction of Compiler and Architecture. In *Computer*, V.30 n.12, December 1997.