Efficacy of Statistical Sampling on Contemporary Workloads: The Case of SPEC CPU2017

Sarabjeet Singh
Ashoka University
sarabjeet.singh@ashoka.edu.ac.in

Manu Awasthi
Ashoka University
manu.awasthi@ashoka.edu.ac.in

***Abstract*—New benchmark suites are constantly being released, with each one providing a much larger set of benchmarks, representing an ever-growing variety of workloads. Contemporary workloads are increasingly more complex in their computational and memory footprints. Most computer architecture research is based on the ability of researchers to simulate novel ideas with a variety of workloads representing the domain being researched. However, bigger and complex benchmarks suites have made it extremely impractical to simulate complete benchmarks from start to finish. As a result, architects are becoming increasingly dependent on statistical sampling techniques like SimPoints, which identify long, repetitive execution phases in benchmarks, and limit simulations to a few instances of these phases. These techniques present an inherent trade-off between simulation speed and accuracy.**

This work presents results and insights for determining the accuracy of simulation points for the SPEC CPU2017 suite, using Pin and PinPoints, which is an implementation of SimPoints for the x86 ISA. Our analysis concludes that carefully chosen simulation points faithfully represent the workload; we observe $<$1% variance in the instruction distribution between full runs and the ones using SimPoints, while reducing simulation time by $\sim$750$\times$. We also show that on average, just 12 phases can faithfully represent the 90^{th} percentile of a benchmark's behavior, which can help reduce the overall simulation time by up to $\sim$1297$\times$. In addition, using performance statistics with native binaries on real hardware and from an architectural model of the same machine using SimPoints, we report good co-relations between the two on metrics such as CPI. Finally, we present cases like memory hierarchy explorations, where SimPoints should be used judiciously and with extreme caution in order to derive correct conclusions – inappropriately chosen SimPoint configurations can show large deviations in memory hierarchy behavior as compared to full runs, as reported by prior studies.

***Index Terms*—Statistical Sampling, Simulation Points, SPEC CPU2017, Workload Characterization**

I. Introduction

Computer architecture exploration is based on the ability of architects to analyze contemporary workloads for bottlenecks and then propose and evaluate architectures to alleviate said bottlenecks. Traditionally, a number of workloads are bundled together, and are provided in the form of standardized benchmark suites. Over the years, a number of such suites have been made available to architects. Of these, the ones being put out by the Standard Performance Evaluation Corporation (SPEC) are arguably the most well known, as well as the most popular ones. They are used for exploratory studies ranging from pipeline optimizations to cache design, all the way to evaluating main memory architectures. SPEC has been releasing a new version of their benchmark suite every few years, since 1992.

In the last decade, computer system architecture as well as characteristics of real world applications have undergone tremendous changes. To keep up with technological advancements and cover the scope of emerging applications, SPEC CPU2017 has very recently been introduced [1]. This suite is a major upgrade over SPEC CPU 2006 [2]: new benchmarks have been added and existing workloads in the suite have been changed to have significantly larger dynamic instruction counts and data footprints than both the earlier versions - CPU2006 and CPU2000 [3].

Increase in the numbers and footprints of workloads in a suite is both a boon and a bane for computer architects, who depend on architectural simulations to evaluate novel ideas. Increase in the dynamic instruction count and working set sizes have led to extremely large run times for simulating contemporary architectures. Modeling future architectures, ones with larger core counts, deeper cache hierarchies and specialized compute elements is bound to increase simulation times even further. As an example, GEM5 and MARSSx86, two very popular architectural simulators in the full system simulation mode, provide a simulation speed of about 200 KIPS. On the other hand, simulators likes ZSim and Sniper, which only simulate a workload's user level behavior are a little faster and can execute a simulated machine at 20 and 2 MIPS, respectively [4], [5].

To keep the simulation times in check, researchers often utilize approaches like statistical sampling of workloads. Sherwood et al. [6] were the first to propose one such technique, termed SimPoints (short for simulation points) for speeding up architectural simulations. SimPoints provide a mechanism for reducing execution time of workloads without incurring significant errors. The technique takes advantage of repetitive phases of execution in a workload, which, when simulated individually, would suffice to mimic the workload's behavior. Furthermore, since the method of identifying long, repetitive phases of a workload is independent of the ISA, phases identified by SimPoints hold valid for any ISA.

In this work, we apply the SimPoints statistical sampling methodology to SPEC CPU2017 and come up with a number of interesting observations. We use these observations to propose a few best practices, which if not followed, might

lead to incorrect conclusions. The main contributions of the paper are enumerated below.

- We carry out a detailed design space sweep for determining possible simulation points/phases for SPEC CPU2017 benchmarks and come up with an optimal number of 35 phases and 30 million instructions per phase being able to represent the behavior of all considered workloads. This detailed sweep is necessary for determining optimized SimPoint configurations that will help reduce some of the discrepancies in metrics like cache miss rates (as compared to full benchmark runs) which have been observed in prior studies [7].
- We show that, using SimPoints, we can simulate the benchmarks ∼750x faster and with ∼650x less instructions, as compared to whole benchmark simulation.
- We show that if SimPoints are well chosen, there exists extremely good correlation between the instruction distributions obtained from native runs of benchmarks and those identified by the SimPoint methodology, with an error rate of less than 1% between the average of two, across the entire benchmark suite.
- We show that there exist inherent trade-offs between simulation speed and simulation accuracy along certain axes, especially memory hierarchy simulation, for which special care needs to be taken, if simulation is done using SimPoints. Variations in results using SimPoints can be reduced if appropriate mitigation techniques (like cache warming before each phase) are used. If this is not done, exploration of memory hierarchies using SimPoints can lead to incorrect design choices.
- We verify the accuracy of simulation of SimPoints driven workloads against performance counter data obtained from real hardware using native execution of workloads, and show that for well chosen SimPoints, there exists good co-relation between the two – the difference in average CPI for a workload between the two methods being 2.59%.
- We show that, on an average across the benchmark suite, 12 representative phases of a SPEC CPU2017 benchmark are able to capture 90% of the workload's overall behavior, leading to greatly reduced simulation speeds - up to ∼1297× as compared to whole benchmark simulation.

The rest of the paper is organized as follows. Section II gives a background of SPEC CPU2017 benchmarks and PinPlay. Section III describes the methodology to create the simulation points of these benchmarks. Section IV discusses the accuracy of characterization of the benchmarks using simulation points. Finally, we discuss the related work in Section V and conclude in Section VI.

II. Background

A. SPEC CPU2017

SPEC CPU is a widely acknowledged suite of compute intensive benchmarks, which tests processor's, memory system's and compiler's performance. A number of versions of SPEC have been released over the years, with the latest version, released in 2017, and aptly named, SPEC CPU2017. CPU2017 [1] considers state-of-the-art applications, organizing 43 benchmarks into four different classes: 10 speed integer (SPECspeed INT), 10 rate integer (SPECrate INT), 10 speed floating point (SPECspeed FP) and 13 rate floating point (SPEC rate FP). The speed and rate suites vary in workload sizes, compile flags and run rules. SPEC*speed* measures the time for completion by running a single copy of each benchmark, with an option of using multiple OpenMP threads. Hence, speed is a measure of single instance performance, typically measured by metrics like IPC (Instructions Per Cycle). On the other hand, SPEC*rate* measures the throughput of the overall chip, with possibly multiple cores, by running multiple, concurrent copies of the *same* benchmark with OpenMP disabled. Most applications have both rate and speed versions (denoted by *_r* and *_s*, respectively), except for *namd*, *parest*, *povray* and *blender*, which only have the rate versions, and *pop2*, which only has the speed version. Similar to SPEC CPU2006, SPEC CPU2017 has been provided with three input sets: **test** (to test if executables are functional), **train** (data for feedback-directed optimization), and **ref** (timed data set).

In the current iteration, many new benchmarks have been included to cover emerging application domains. In INT category, artificial intelligence (AI) has been extensively represented with three new benchmarks, namely *deepsjeng* (alpha-beta tree search & pattern recognition), *leela* (Monte Carlo tree search, game tree search & pattern recognition) and *exchange2* (recursive solution generator). Two data compression benchmarks, *xz* (general data compression) and *x264* (video compression), have been also added. In the FP category, nine new benchmarks have been added: *parest* implements a finite element solver for biomedical imaging; *blender* performs 3D rendering; *cam4* (atmosphere general circulation modeling), *pop2* (climate modeling) and *roms* (regional ocean modeling) represent the climatology domain; *imagick* is an image manipulation application; *nab* is a floating-point intensive molecular modeling application representing the life sciences domain; *fotonik3d* (computational electromagnetics) and *cactuBSSN* (general relativity) represents the physics domain.

B. Pin and PinPlay

a) Pin: Pin [8] is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction sets. It provides a rich set of APIs that can be used to study various characteristics of program behavior at the ISA level. Since Pin dynamically instruments programs, a technique which inserts extra code into an existing application to collect run time information and runs at native execution speeds, it provides orders of magnitude of speedup over a functional simulator. Using APIs provided by Pin, multiple tools have been created for studying multiple characteristics of different programs. Example of such tools include *inscount0* (dynamic instruction counter), *ldstmix* (dynamic register/memory operand pattern profiler), *allcache* (functional simulator of instruction+data TLB+cache hierarchies), *logger* (records execution traces)

and *replayer* (replays the logged execution traces). Most of these tools, which are shipped as a part of the Pin software distribution, were used actively in this study.

b) Pinballs: Pinball [9] is a user-level checkpoint format of a program's execution, which can be loaded and executed to recreate that particular execution. A Pinball is created and consumed using a Pin based framework called PinPlay [10]. PinPlay consists of two Pintools: (i) a *logger* that captures the initial architectural state and non-deterministic events during a program's execution in a set of files which are collectively called a *Pinball*; and (ii) a *replayer* that runs a Pinball, repeating the captured program's execution. The replayer can be executed with Pintools dynamically instrumenting the Pinball or can be integrated with a Pin-based simulator allowing simulations based on Pinball(s) instead of a binary. A Pinball can be created for the execution of the entire program or for a part of execution. A large Pinball of whole execution can be broken down into a number of smaller *regional* Pinballs. This allows users to work with (run or simulate) either the entire workload, or a portion thereof.

Using Pinballs instead of complete binaries has the advantages of being independent of the operating system, reduction in data+code footprint as compared to native binary and data sets, as well as reduction in non-determinism of the program. Also, since Pinballs are self-contained, to include all the information that is needed for the program's execution, the program binary, input files and special licenses are not needed during replay. In general, the logging is 100-200× slower [11] as compared to native execution, which results in extremely long times for creating them. However, the increased ease of portability, once Pinballs have been captured, outweighs the one time overheads of creating the Pinballs.

c) PinPoints: Pin + SimPoints: Modern processors are highly intricate and detailed; cycle-accurate simulation of such systems is extremely slow, hence computer architects depend on simulation methodologies that use statistical sampling to capture program behaviors. In this work, we use *SimPoints* [12], a mechanism of statistically sampling similar phases in an execution to remove redundancies in execution behavior, resulting in reduced simulation times. A typical program's behavior follows a number of long repetitive behaviors, called *phases* [13]. SimPoints provides a means of identifying and isolating these unique slices (or phases) by dynamically slicing the execution trace into smaller, equal sized chunks and grouping similar slices together. Grouping is done by creating Basic Block Vectors [14] (BBVs) of each slice and forming a cluster of BBVs which are close to each other. The degree of closeness is determined by a set of architectural metrics, namely performance, branch misprediction, cache misses, etc [15]. Since the behavior of the program at a given time is directly related to the code executed during that interval, slices grouped under the same cluster are expected to have similar behaviour [14], [16]. K-means clustering [17] is employed to form clusters of similar slices. A number of slices may exist within the cluster. However the slice closest to the average behaviour of that cluster is chosen as the cluster's

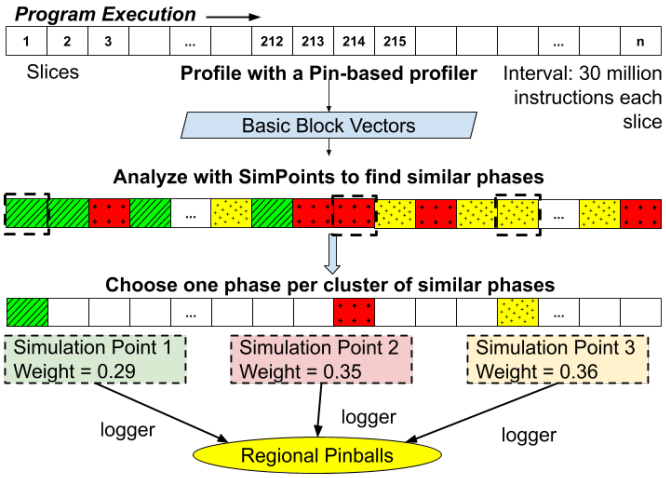


Fig. 1. SimPoints procedure

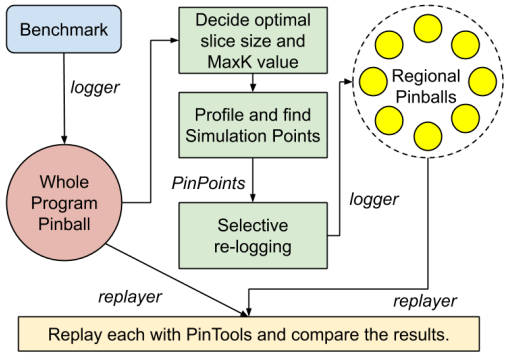


Fig. 2. SimPoints methodology using PinPoints

representative phase and is called a *simulation point*. The number of slices in each cluster determines its *weight* which represents the contribution of that simulation point towards the whole program. Higher weight would imply that the phase gets executed more frequently. Inclusion of all simulation points, along with their weights may be used to predict the behavior of the program. This procedure is depicted in Figure 1. Accuracy of SimPoints methodology depends on many factors, two of the most important ones being the number of instructions in the slices, also known as *slice length*, and *MaxK* value of the clustering algorithm, which represents the maximum number of clusters. Large slices may fail to capture certain short-lived phase behaviors, but can generate reasonably accurate results without the need to warm up the caches. Reduction in the slice length makes each simulation point susceptible to errors due to cold cache misses, but will identify, possibly more phases of shorter durations, leading to better simulation accuracy. Similarly, a large enough MaxK value needs to be selected to avoid compromising the selection of simulation points.

PinPoints [18] use SimPoints with Pin to analyze the dynamic instruction trace of the program and identify simulation points. In this work, we create checkpoints of program execution (Pinballs) using PinPlay's logger tool, and generate

TABLE I
ALLCACHE SIMULATOR CONFIGURATION

L1i	32-way, 32kB, 32B linesize
L1d	32-way, 32kB, 32B linesize
L2	Unified 2MB direct-mapped, 32B linesize
L3	Unified 16MB direct-mapped, 32B linesize

checkpoint of whole benchmark execution **(Whole Pinballs)**, which are then used as inputs to PinPoint for generation of checkpoints of simulation points **(Regional Pinballs)** and their weights. The entire flow of the experimental methodology followed in this work is depicted in Figure 2. The Regional Pinballs can be directly run with the replayer pintool to report statistics.

III. METHODOLOGY

SPEC CPU2017 benchmarks are compiled using gcc compiler with SPEC recommended optimization flags, for reference input size. Speed workloads are compiled to use 4 OpenMP threads, while the rate workloads were executed with a single instance of the benchmark. The experimental methodology for identifying simulation points and creating checkpoints or Pinballs is depicted in Figure 2. As observed from the figure, the compiled binaries are used to create Whole Pinballs, using PinPlay's logger tool. This is an extremely slow process and results in a 100-200× execution slowdown, as also reported in previous studies [11]. Some benchmarks of the suite, especially ones belonging to Floating Point subsuite, couldn't complete this process in a very long time. For example, on a Intel(R) Xeon(R) CPU E5-2650, 2.00GHz, with 32kB, 8-way L1I/L1D caches, 256kB 8-way L2 cache, and a unified, 20MB 20-way L3 cache, checkpointing *bwaves_s* took more than a month of computation time. As a result, in this work, we present a subset of the complete benchmark suite, and keep the rest for future work. Finally, we use the Whole Pinballs to create Regional Pinballs, as mentioned in previous sections. With the discussion about tools, we now present some of the insights that we have gathered with multiple types of experiments.

IV. RESULTS AND DISCUSSION

A. MaxK and Slice Size

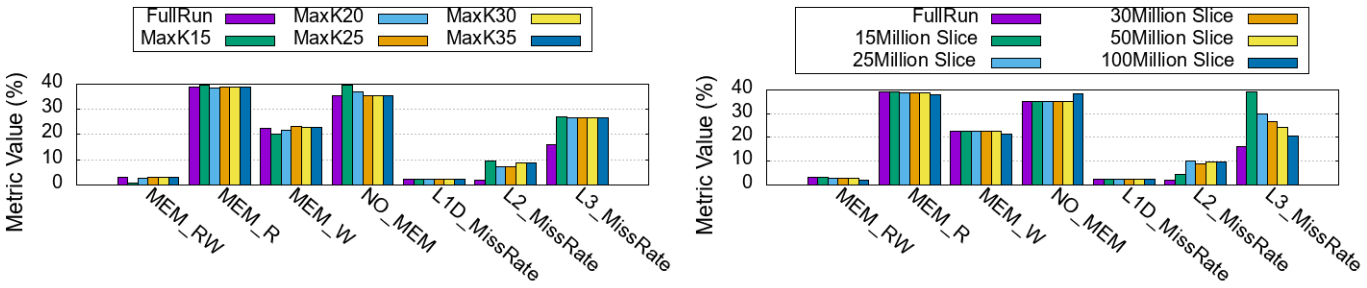
Since the accuracy of simulation points and hence the characterization of CPU2017 depends largely on the number of clusters of phases that are allowed (value of MaxK), and slice size, which is the number of instructions per phase, we create simulation points for varying values of MaxK and slice sizes while keeping the other one constant. We then compare the two in their ability to match results of complete runs. These experiments are used as the basis for deciding the optimal values to profile benchmarks using PinPoints for the rest of this paper. In addition, this sweep helps explore the various tradeoffs associated with choosing different values of the two variables. Figures 3(a) and 3(b) present the results of the design space sweep for MaxK and slice sizes for one benchmark, *xalancbmk_s*. Due to lack of space, we are unable to provide experimental data from all considered benchmarks. However, the conclusions derived from this data point remain largely consistent across all the benchmarks under consideration.

TABLE II
SPEC CPU2017 SIMULATION POINTS

Benchmark	**Number of Simulation Points**	**Number of 90 percentile Simulation Points**
500.perlbench_r	18	11
502.gcc_r	27	15
505.mcf_r	18	9
520.omnetpp_r	4	3
525.x264_r	23	15
531.deepsjeng_r	20	15
541.leela_r	19	12
548.exchange2_r	21	16
557.xz_r	13	7
600.perlbench_s	21	13
602.gcc_s	15	5
605.mcf_s	28	14
620.omnetpp_s	3	2
623.xalancbmk_s	25	19
625.x264_s	19	13
631.deepsjeng_s	12	10
641.leela_s	20	13
648.exchange2_s	19	15
657.xz_s	18	10
503.bwaves_r	26	7
507.cactuBSSN_r	25	4
508.namd_r	26	17
510.parest_r	23	14
511.povray_r	23	19
519.lbm_r	22	8
526.blender_r	22	14
538.imagick_r	14	7
544.nab_r	22	10
549.fotonik3d_r	27	11
Average	19.75	11.31

We compared different values of MaxK and slice size using two main metrics - instruction mix and cache miss rates for an arbitrary cache hierarchy, defined in Table I. These metrics were calculated using the *ldstmix* pintool and the *allcache* simulator, respectively. The instructions are broken down into four different categories - memory reads (MEM_R), memory writes (MEM_W), memory read and writes (MEM_RW)[1] and everything else (NO_MEM). The MaxK value was varied from 15 to 35 in steps of 5. We experimented with slice sizes of 15, 25, 30, 50 and 100 million instructions. These runs were then compared against the full run (without statistical sampling). Figure 3(a) provides a relative comparison between these experiments across metrics under consideration while varying MaxK, whereas Figure 3(b) provides the same for varying slice sizes, while keeping the MaxK constant at 35. As can be observed, for smaller values of MaxK, there are significant variations between the instruction distributions as compared to the full run. Since a benchmark can have many representative phases and by selecting a small MaxK, we

[1]Memory-to-memory instructions like *movs* in x86 account for MEM_RW instructions.



(a) Accuracy while varying MaxK, for *xalancbmk_s*.

(b) Accuracy while varying Slice size, for *xalancbmk_s*.

Fig. 3. Sensitivity analysis of MaxK and Slice Size, depicted for *xalancbmk_s*.

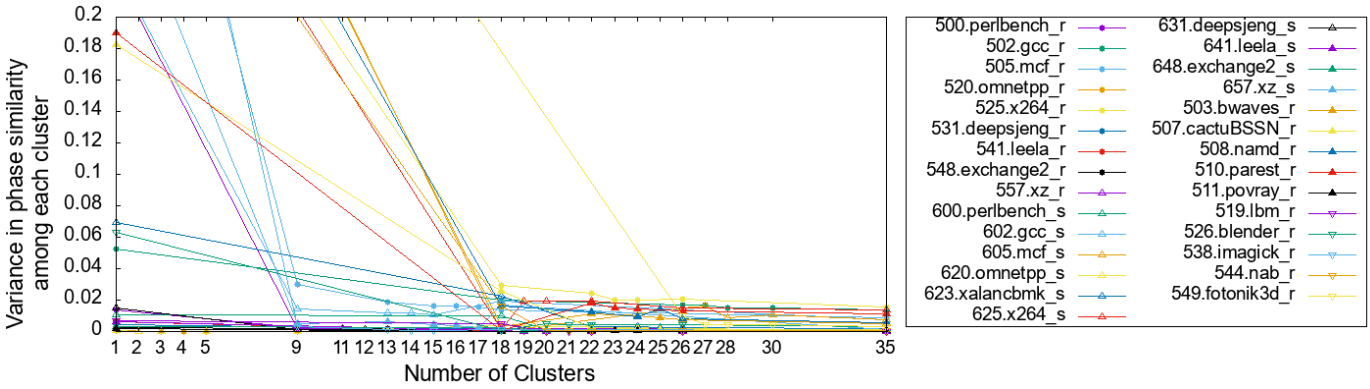


Fig. 4. Average Variance in phase similarity among each cluster for varying number of clusters, depicted for each benchmark.

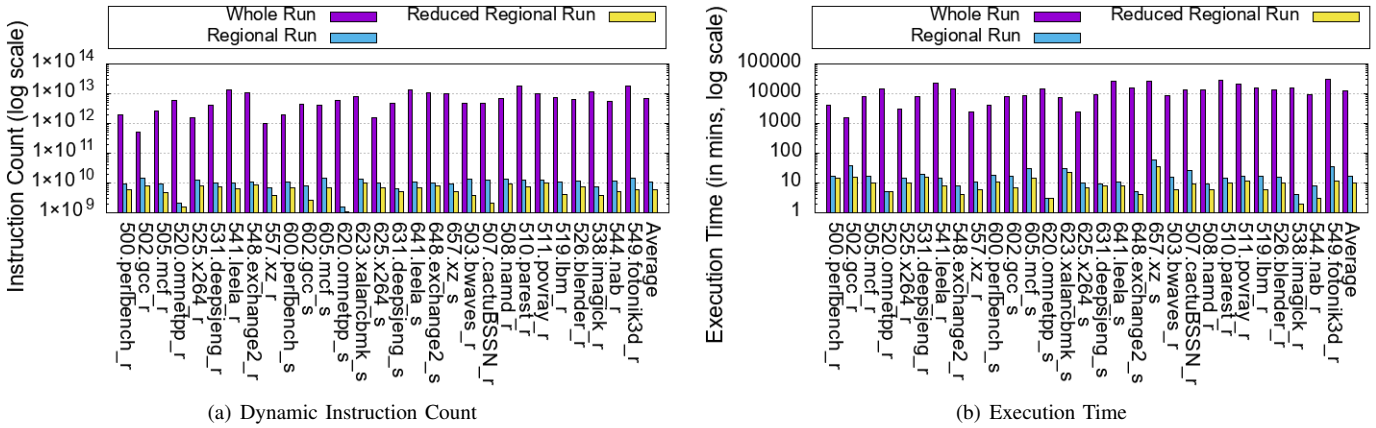
force the sampling mechanism to compromise its selection of representative points. This results in larger variations. We observe that most SPEC CPU2017 benchmarks, do not need more than 35 clusters to capture all the phases of the benchmark. Table II compiles the results of these experiments - all benchmarks have number of simulation points that are well below the chosen maximum value of 35.

As discussed in Section II, a number of phases may exist in a cluster but the one with behaviour closest to the average behaviour of all phases in that cluster, is chosen as the simulation point. However, phases deviating from this average behaviour might not be perfectly represented by the respective simulation point. A measure of how far this deviation stretches, termed variance, also determines the number of simulation points that are selected. Forcing a low number of clusters reduces the number of unique behaviours that can be captured. This phenomenon is illustrated in Figure 4, where we show the average variance in similarity among the cluster's phase behaviors, for most of the benchmarks from the suite. As expected, it can be seen that as number of available clusters decrease, the phases try to adjust themselves within these clusters at the expense of accuracy.

With the MaxK value at 35, we sweep across multiple values of slice lengths. In Figure 3(b), we observe that small slice sizes have very large deviations from the full run, especially with respect to cache miss rates, even though the percentage distribution of memory instructions doesn't change much. We believe that this is caused by the reduced number of overall memory accesses in the statistical samples, leading to increased cold cache effects. As a result, there is a large variation in the miss rates of caches that are further away from the processor. This is confirmed by the fact that increasing the slice length leads to dramatic reductions in the L3 cache miss rates, bringing them much closer to the ones for full run. We select 30 million slice size, which might augment the cold cache effect, but also helps capture the effect of short-lived phases; hence contributing to the accuracy.

B. Whole v/s Regional Pinball Run

Next, building upon the optimal values of MaxK and slice size derived in Section IV-A, we profile the benchmarks using PinPoints. PinPoints generates the checkpoints (also referred to as Pinballs) of simulation points and their respective weights. As was described previously in Section II, these checkpoints are termed as *Regional Pinballs*, and their execution as ***Regional Run***. Figure 5(a) & 5(b) depict the dynamic instruction count and execution time, respectively, of Whole and Regional Runs of the benchmark suite. PinPoints methodology is able to create checkpoints of the SPEC CPU2017 benchmark suite with $\sim$650$\times$ reduction in the number of executed instructions. Across the benchmark suite, on an average, the number of executed instructions are reduced from 6873.9 billion to 10.4 billion. Consequently, the average execution time of single benchmark dropped from **213.2 hours** for the Whole Run to



(a) Dynamic Instruction Count

(b) Execution Time

Fig. 5. Comparison of Whole Run, Regional Run and Reduced Regional Run in terms of Dynamic Instruction Count and Execution Time.

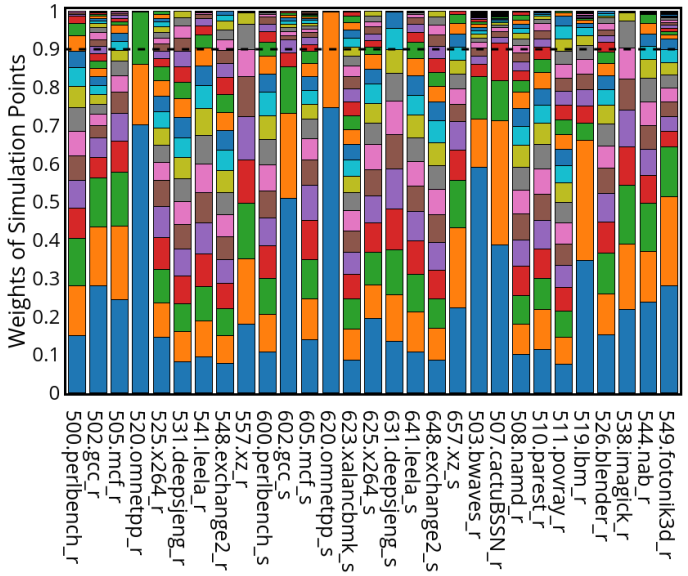


Fig. 6. Weight of each simulation point of the benchmarks.

17.17 minutes for the Regional Run – a ∼750× reduction in run time. Architectural experiments generally require longer runs demanding a significant amount of time. Using Regional Pinballs can provide a solution by offering significantly reduced simulation time.

C. 90^{th} Percentile Simulation Points

While programs may have a large number of phases, many have only a few dominant ones. We analyze the weights associated with the phases / simulation points [2] to get insights into the nature of phase behavior. In Figure 6, we present the weights associated with different phases in the workload. Each stacked bar in the figure represents a benchmark, divided into sub-bars representing its simulation points. The height of each of the sub-bars represents the weight of a particular phase. The number of sub-bars represent the number of phases in the benchmark. Most programs have less than 25 overall simulation points, with only 5 programs exceeding that value. This observation is also tabulated in the second column of Table II. A higher weight of a simulation point would imply that the program spends a larger fraction of time in executing that phase. For example, *503.bwaves_r* has one dominant simulation point that accounts for 60% of the overall execution, and three highest weighted simulation points, together account for 80% of the program's execution, indicating a low diversity in benchmark behavior. On the other hand, benchmarks like *631.deepsjeng_s*, *648.exchange2_s* and *511.povray_r* have a fairly regular distribution of weights, and would need more number of simulation points for accuracy. Certain benchmarks like *503.bwaves_r*, *507.cactuBSSN_r*, *519.lbm_r* have many simulation points with almost insignificant weights. While their existence indicates some diversity in the benchmark's behavior, they do not significantly contribute to the overall execution profile. Hence, such benchmarks can be simulated accurately with a small number of simulation points. To verify this hypothesis, we compare the total number of simulation points to the number of simulation points contributing to 90% of the execution. This is done by sorting the simulation points in the descending order of their weights, and then selecting them until the total sum of weights adds up to 0.9 (out of a total of 1.0). This is pictorially represented by a dashed line in Figure 6. As a result of this optimization, the average number of simulation points for a benchmark drops from 20 to 12, as depicted in the third column of Table II. We term execution these simulation points as ***Reduced Regional Runs***. Considering only the 90^{th} percentile simulation points has a huge impact on run times; reduction of dynamic instruction count by 1225× and simulation time by 1297× as compared to Whole Run, as depicted in Figure 5. Across the entire suite, compared to Regional Runs, Reduced Regional Runs execute 1.743× less instructions and reduced simulation time by 1.741×. Similarly, one can further reduce the number of simulation points leading to smaller execution times. Since many simulation points have insignificant weights, they can

[2]We use phases and simulation points interchangeably in the rest of this discussion.

be eliminated with a little trade-off in accuracy.

D. Comparison of Whole, Regional and Reduced Regional Runs

Next, we determine the accuracy of simulation points in representing the benchmark by replaying Whole, Regional and Reduced Regional Pinballs using Pintools for profiling. These experiments aim to compare the ability of the Regional and Reduced Regional Pinballs to represent the original workload. As before, we compare these three along two main axes - instruction distribution and cache miss rates. We use *ldstmix* and *allcache* Pintools for these set of experiments, which report instruction distributions and cache miss rates, respectively. The configuration of cache hierarchy simulated by *allcache* is provided in Table I.

The PinPoints methodology divides the entire execution into multiple, smaller executions by pointing out simulation points (Regional Pinballs). The way these simulation points are executed can report variable statistics. In this study we adopt the methodology described in [11], [19], where each Regional Pinball is executed individually with Pintools, and a weighted average of the statistics reported by each is finally reported. The weight of a simulation point is the number of times the phase repeats itself in its cluster, during the full execution, divided by the total number of phases in all the clusters. Since each Pinball can be executed independently, these are executed in parallel to save time. It is also important to note that the weighted average should be taken only for statistics normalized by instructions; CPI computation is allowed, but IPC is not.

Instruction profile for the floating point and integer benchmarks is distributed into four categories: instructions that do not refer memory (NO_MEM), instructions that have one or more source operand in the memory (MEM_R), instructions whose the destination operand is in memory (MEM_W), and instructions whose source and destination operands are in memory (MEM_RW). Figure 7 provides the instruction distributions for all the three cases. As can be observed, the percent distributions for each category almost match up perfectly with that of the Whole Run, which, on average has 49.1% compute-only instructions, 36.7% memory-read and 12.9% memory write instructions. As compared to the Whole Runs, the errors for both the Regional Runs, as well as the Reduced Regional Runs are less than 1%. Furthermore, the Regional Runs executed on the cache simulator, exhibit some variation in the cache miss rates as compared to the Whole Runs. The average L1D, L2 and L3 cache miss rates are 0.18%, 0.10%, and 25.16% higher respectively, than that of the Whole Runs, as shown in Figure 8(a)-(c). L1I has negligible miss rates in all cases, and does not affect the methodology's accuracy in sampling the workloads. The errors in cache miss rates for the Reduced Regional Runs are very close to those observed for Regional Runs. As compared to the Whole Run, we observe that average cache miss rates of the Reduced Regional Run increased by 2.23%, 0.33%, and 25.53% for L1D, L2 and L3 caches respectively. Therefore, by simulating just 12 phases, we can represent the benchmark behavior with a little trade-off in memory hierarchy simulation accuracy. As opposed to existing work [7] in determining simulation points for SPEC CPU2017, which show >30% variation in L1D miss rate prediction, we show that carefully chosen simulation points can generate more accurate results.

As discussed in Section IV-C, while applications may have a large number of phases, only a few of dominant ones among them contribute significantly to the overall execution profile. Hence, the other simulation points can potentially be removed with little inaccuracy, saving execution time. In Figures 5, 7 and 8, we illustrated this trade-off considering simulation points contributing to 90% of the total weight. However, one can further reduce the simulation points to achieve smaller execution times. In Figure 9, we provide a design sweep analysis of inaccuracies incurred while reducing the number of simulation points. On y1-axis, we show errors in the considered metrics compared to the Whole Run, averaged across the benchmark suite. On y2-axis, we depict the execution time, while varying the percentile of simulation points considered for execution on x-axis. As expected, the experiment shows that as we reduce the number of simulation points, the error rates go up. Using this analysis, one can judicially choose the number of simulation points for execution based on their accuracy/runtime budget.

Apart from LLC miss rates, the PinPoints methodology is able to maintain the cache behavior. The discrepancy in the LLC miss rates between the three types of runs is due to the reduced number of L3 accesses – the overall number of instructions executed by Regional and Reduced Regional Pinballs is smaller as compared to Whole Pinballs. This is clearly observed in Figure 10, which shows the reduced number of accesses to L3 cache in the Regional and Reduced Regional Runs, leading to discrepancies in cache miss rates. As a result, for memory accesses, Regional or Reduced Regional Runs might not be the most optimal simulation platform. To alleviate this, the set of Regional Pinballs must be run multiple times, thus exercising the LLC to remove the cold cache effects, or else a larger slice length should be chosen. To verify this hypothesis, we run each simulation point, with slice size of 30 million, and allow the caches to be warmed up for 500 million cycles before every simulation point starts executing. We model the cache hierarchy used in Table I in Sniper [20] to simulate this experiment. We present these results in Figure 8 as ***Warmup Regional Run***. We observe that after alleviating cold cache effect, the error in average LLC miss rate go down drastically, from 25.16% to 9.08%. Therefore, we conclude that Regional Pinballs, executed in proportion to their weights with reasonable warmup, are capable of representing the similar behavior of the whole benchmark. However, studies not taking into account these subtle experimental details are bound to make inaccurate conclusions.

E. Native Execution v/s Sniper with Simulation Points

Next, we extend the comparison of Whole, Regional and Reduced Regional Runs for use of a system simulator. We

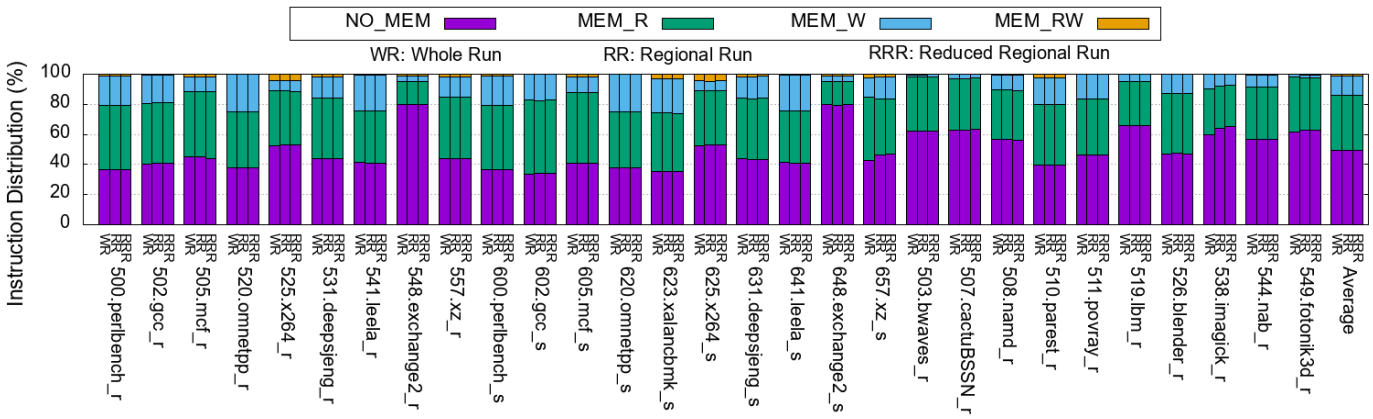
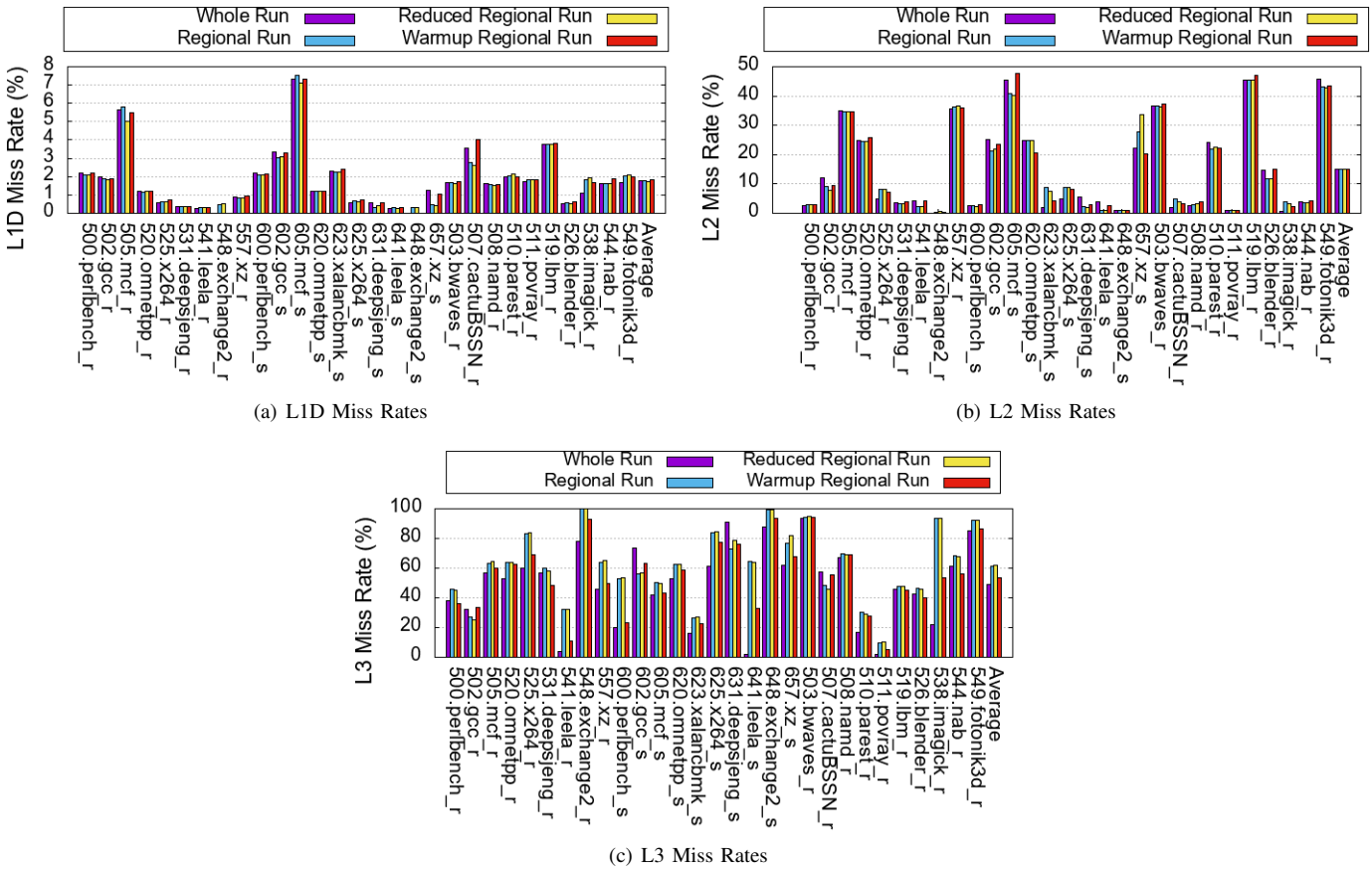


Fig. 7. Instruction Distribution Comparison of Whole Run, Regional Run and Reduced Regional Run.



(a) L1D Miss Rates

(b) L2 Miss Rates

(c) L3 Miss Rates

Fig. 8. Cache Miss Rates Comparison of Whole Run, Regional Run, Reduced Regional Run and Warmup Regional Run, for configurations in Table I.

aim to compare the accuracy of using various mechanisms of statistical sampling of benchmarks and compare them to results obtained from real hardware, using perf tool [21]. First, we run the benchmarks on real hardware - an 3.4GHz, 8-core Intel i7-3770 with 8 GB RAM and record the most relevant statistics using native binaries. Then we use an architectural simulator, Sniper [20], to model the real hardware as faithfully as possible. Table III presents the configuration of the simulated machine. We then run the Regional and the Reduced Regional Pinballs within the simulated machine. Finally, we compare all three to check for inaccuracies that might be introduced by statistical sampling mechanisms. Since many of the perf's symbolic event were not supported on the processor we had access to, we limit our comparison to CPI, which is calculated as the ratio of the hardware events *cpu-cycles* and *instructions*. We simulate Regional and Reduced Regional Pinballs within the simulator to capture CPI information. Congruence of these two results would signify the accuracy

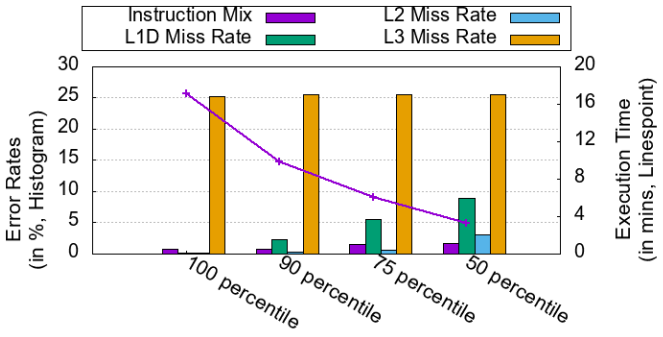


Fig. 9. Error rates of metrics, compared to Whole Run *(y1-axis, Histogram)*. Execution time *(y2-axis, Linespoint)*. Results shown are averaged across the benchmark suite. 100 and 90 percentile executions represent Regional and Reduced Regional Runs.

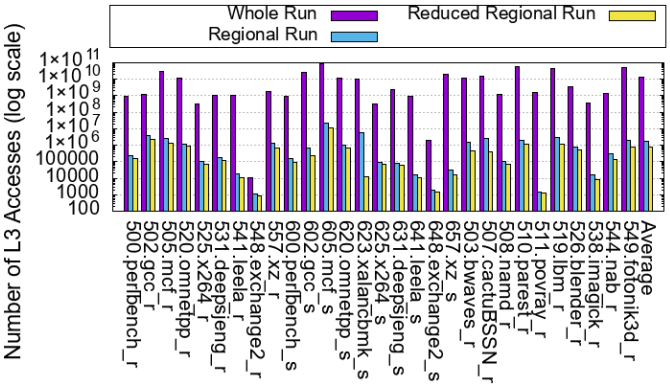


Fig. 10. Number of L3 cache accesses by Whole Run v/s Regional Run v/s Reduced Regional Run, for configuration in Table I.

TABLE III
SYSTEM CONFIGURATION

Model	8-core Intel i7-3770
CPU Frequency	3.4GHz
Pipeline	19 stage Out-of-Order
Fetch Width	6 instructions per cycle
Decode Width	4-7 fused μ-ops per cycle
Rename width and Issue width	4 fused μ-ops per cycle
Dispatch width	6 μ-ops per cycle
Commit width	4 fused μ-ops per cycle
Reorder buffer	168 entries
Branch Reorder Buffer	48 entries
Branch misprediction penalty	8 cycles
L1-I cache & latency	32 KB, 8-way & 4 cycles
L1-D cache & latency	32 KB, 8-way & 4 cycles
L2 cache & latency	256 KB, 8-way & 10 cycles
L3 cache & latency	8 MB, 16-way & 30 cycles
Cache line size	64 Bytes

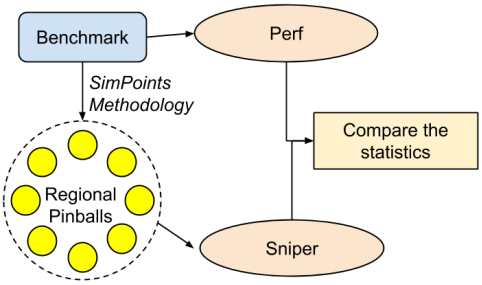


Fig. 11. Benchmark Execution: Perf v/s Sniper

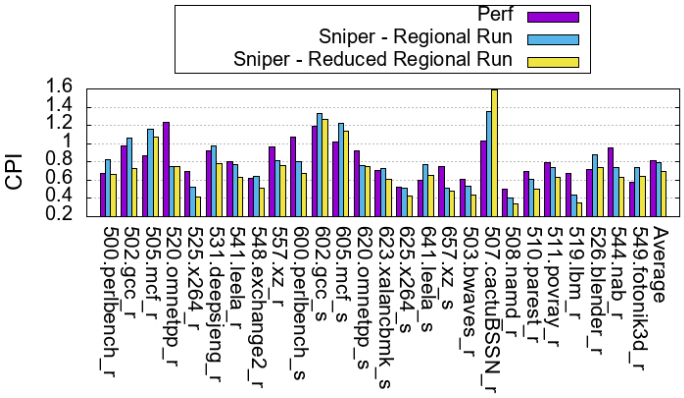


Fig. 12. CPI of SPEC CPU2017, running natively versus running on Sniper, for configuration in Table III.

of characterization of SPEC CPU2017 using the PinPoints methodology on Sniper, to that of a real system execution. The procedure is depicted in Figure 11.

We compare the CPI values of benchmarks in both executions in Figure 12. It should be noted that these result also include errors due to non-determinism and execution on Sniper. The CPI for Regional Pinballs within Sniper correlates well with that of native execution - an error of 2.59% in average CPI across all benchmarks was observed. Next, we validated the accuracy of Reduced Regional Pinballs. As can be seen from Figure 12, few programs like *507.cactusBSSN_r* are outliers, whose deviation in Regional Runs, as compared to native execution is extremely large. This divergence increases for the Reduced Regional Runs. However, for most benchmarks, the deviation is not very high. Across the entire suite, as compared to the Whole Run, an average deviation of 13.9% was observed in Reduced Regional Runs.

V. RELATED WORK

A number of previous studies have generated simulation points for a variety of benchmark suites. However, to the best of our knowledge, this paper presents the first systematic study of the efficacy of applying the SimPoints methodology to the SPEC CPU2017 suite.

A. SPEC CPU 2017 Characterization

Limaye and Adegbija [22] use hardware performance counter statistics to characterize SPEC CPU2017 applications

with respect to several metrics such as instruction distribution, execution performance, branch and cache behaviors. They also utilize Principal Components Analysis [23] and hierarchical clustering to identify subsets of the suite. Similarly, Panda et al. [24] also characterize CPU2017 Speed benchmarks using *perf*, and leverage statistical techniques to identify cross application redundancies and propose subsets of the entire suite, by classifying multiple benchmarks with similar behaviors into a single subset. Further, they also provide a detailed evaluation of the representativeness of the subsets. Bucek et al. [25] present an overview of CPU2017 suite and discuss its reportable execution. Wu et al. [7] study the phase behaviour of the SPEC CPU2017 suite and have publicly released simulation points for many CPU2017 workloads, as a part of their research efforts. Further, they have also studied time-varying application behaviour and correlated it with the simulation points being considered during those phases. In contrast, the goal of this work is to evaluate the efficiency of sampling using simulation points for the SPEC CPU2017 suite. We carry out a detailed design space exploration for determining best possible simulation points for the benchmarks. We acknowledge that the benchmark behaviour is a function of the algorithms being implemented, and could be a complete study in itself. We refer to [3] for benchmark behavior which presents a detailed, memory-centric analysis of workloads from the SPEC CPU2017 suite.

B. Statistical Sampling Techniques

Joshua et al. [26] have evaluated the accuracy and coverage of the most promising benchmark subsetting approaches, including principal component analysis (PCA), k-means clustering, performance bottlenecks, memory characteristics and instruction distributions, using SPEC CPU2000 benchmarks. Eeckhout et al. [27] compare accuracy of SimPoints sampling approach against the technique of selecting representative benchmarks of a suite as subsets. Shaccour and Mansour [28] propose a loop-centric methodology that targets loop dominant programs by exploiting internal program characteristics to reduce cross program computational redundancies. Wenisch et al. [29] proposed SimFlex, which enables parallelism in statistical sampling, reducing sampling time. They also discuss overcoming practical constraints imposed by simulators, such as fast-forwarding between the measurements and warming to eliminate cold-start biases. Hamerly et al. [30] extend SimPoints to support variable slice lengths, increasing representativeness of simulation points. Amaral et al. [31] propose the Alberta Workloads for the SPEC CPU2017 benchmark suite in hope to improve the performance evaluation of techniques that rely on any type of learning, for example the formal Feedback-Directed Optimization (FDO). Nair and John [32] were the first to evaluate and validate simulation points for SPEC CPU2006 and CPU2000 as a method of simulation acceleration. Interestingly, we note that the average number of simulation points of the SPEC CPU suites has not varied significantly through the years, including the CPU2017, as reported in Table II. These results suggest that even though there has been orders of magnitude increase in the dynamic instruction count and the number of memory accesses, the benchmark suites still are constituted using the same number of phases. Sandberg et al. [33] propose a simulation methodology, Full Speed Ahead, (FSA) which aims to minimize the overhead of fast-forwarding by using virtualization to fast-forward the application between the simulation points at near-native speed. As an alternative, Nikoleris et al. [34] uses statistical cache modeling and the workload's memory reuse information (MRI) [35] to eliminate large cache warming periods.

VI. Conclusion

In this work, we evaluate the accuracy of characterizing the SPEC CPU2017 benchmarks using SimPoints methodology. Our analysis shows that architectural simulations using SimPoints, with a maximum cluster size of 35 and slice size of 30 million instructions, have very similar instruction distribution characteristics as Whole Runs – the distribution error is less than 1%. Using SimPoints, we can reduce the simulation time by almost $\sim$750x. However, owing to reduced number of memory instructions and injudiciously chosen SimPoints, significant error rates can be observed in cache miss rates for simulation done using simulation points, which increase for caches further away from the CPU. We show that these can be alleviated by using appropriate mitigation techniques like cache warming and judiciously choosing cluster and slice sizes after carrying out a comprehensive exploration of the available options.

We explore the phase behavior of workloads to find the existence of large, dominant phases which sufficiently represent the benchmark. We take advantage of this nature to reduce the number of simulation points to only those contributing to the 90% of the execution. This results in just 12 simulation points, on average across the suite, representing the benchmarks with some trade-off in accuracy, but with 1225$\times$ reduction in dynamic instruction count and hence, 1297$\times$ reduction in simulation time. Finally, we compare performance of whole benchmark execution on native hardware using performance counters with that of a detailed architectural model of the same machine using SimPoints. Between the two setups, we observe an error of 2.59% in average CPI across the benchmark suite.

References

[1] "SPEC CPU2017 Documentation," https://www.spec.org/cpu2017/Docs, 2017.

[2] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[3] S. Singh and M. Awasthi, "Memory Centric Characterization and Analysis of SPEC CPU2017 Suite," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ACM, 2019, pp. 285–292.

[4] W. Heirman, T. E. Carlson, and K. Craeynest, "Sniper tutorial at IISWC2013: The Sniper Multi-Core Simulator," http://snipersim.org/w/Tutorial:IISWC_2013_Sniper, 2013.

[5] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer architecture news*, vol. 41. ACM, 2013, pp. 475–486.

[6] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.

[7] Q. Wu, S. Flolid, S. Song, J. Deng, and L. K. John, "Hot Regions in SPEC CPU2017," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," vol. 40, no. 6, pp. 190–200, 2005.

[9] H. Patil and T. E. Carlson, "Pinballs: portable and shareable user-level checkpoints for reproducible analysis and simulation," in *Proceedings of the Workshop on Reproducible Research Methodologies (REPRODUCE)*, 2014.

[10] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 2–11.

[11] H. Patil and M. Chabbi, "PinPlay Tutorial at PLDI2015: Using PinPlay for Reproducible Analysis and Replay Debugging," https://pldi15.sigplan.org/details/pldi2015-workshops/6/PINPLAY-Using-PinPlay-for-Reproducible-Analysis-and-Replay-Debugging, 2015.

[12] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, 2003.

[13] T. Sherwood and B. Calder, "Time varying behavior of programs," *In UC San Diego*, 1999.

[14] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2001, pp. 3–14.

[15] C. Pereira, J. Lau, B. Calder, and R. Gupta, "Dynamic phase analysis for cycle-close trace generation," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2005, pp. 321–326.

[16] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder, "The strong correlation between code signatures and performance," in *Proceedings of ISPASS*, 2005.

[17] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[18] 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 *Proceedings of MICRO*, 2004.

[19] H. Patil and T. E. Carlson, "HPCA tutorial: Deterministic PinPoints: Representative and repeatable simulation region selection with PinPlay and Sniper," https://snipersim.org/w/TutorialHPCA_2013_PinPoints, 2013.

[20] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of SuperComputing (SC)*, 2011.

[21] A. C. De Melo, "The new linux perf tools," in *Slides from Linux Kongress*, vol. 18, 2010.

[22] A. Limaye and T. Adegbija, "A Workload Characterization of the SPEC CPU2017 Benchmark Suite," in *Proceedings of ISPASS*, 2018.

[23] G. H. Dunteman, *Principal components analysis*. Sage, 1989, no. 69.

[24] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?" in *Proceedings of HPCA*, 2018.

[25] J. Bucek, K.-D. Lange *et al.*, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 41–42.

[26] J. Y. Joshua, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John, "Evaluating benchmark subsetting approaches," in *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 93–104.

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

[28] E. M. Shaccour and M. M. Mansour, "A Loop-Based Methodology for Reducing Computational Redundancy in Workload Sets," *IEEE Access*, vol. 6, pp. 9570–9584, 2018.

[29] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[30] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[31] J. N. Amaral, E. Borin, D. R. Ashley, C. Benedicto, E. Colp, J. H. S. Hoffmam, M. Karpoff, E. Ochoa, M. Redshaw, and R. E. Rodrigues, "The alberta workloads for the SPEC CPU 2017 benchmark suite," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 159–168.

[32] A. A. Nair and L. K. John, "Simulation points for SPEC CPU 2006," in *Proceedings of ICCD*, 2008.

[33] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 183–192.

[34] N. Nikoleris, A. Sandberg, E. Hagersten, and T. E. Carlson, "CoolSim: Statistical techniques to replace cache warming with efficient, virtualized profiling," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2016, pp. 106–115.

[35] N. Nikoleris, D. Eklov, and E. Hagersten, "Extending statistical cache models to support detailed pipeline simulators," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 86–95.