SPECcast: A Methodology for Fast Performance Evaluation with SPEC CPU 2017 Multiprogrammed Workloads

ABSTRACT

Performance comparison is a key task in computer architecture research. These evaluations might need to consider the scenario where resources are shared concurrently by a wide range of application classes. In many cases, well known benchmarking tools, such as *SpecCPU* do not provide evaluation metrics under such usage circumstances. Previous attempts to fill the gap with realistic workloads have reiled on random combination of applications, formulating performance comparison as a statistical task to reduce the population size. The computational cost of these approaches is substantial, given the large mix required to achieve statistically meaningful results. In this paper, we present *SPECcast*, a methodology for the *SPEC CPU2017* suite, which can circumvent this issue. The idea relies on exploiting the inner application characteristics to minimize the computational cost without degrading the statistical significance of the results. Using manual source-code annotation, we determine a small portion of each application, denoted Region of Interest (ROI), that accurately resembles the whole program's characteristics. Then, we develop synchronization mechanisms that can concurrently run any combination of applications in the cores of the system. This enables us to run multiprogrammed *SPEC* workloads ~95% faster without losing statistical significance.

A detailed validation of the proposed methodology will be performed and three different use cases for the methodology will be described: Fast performance evaluation of micro-architectural features (prefetching in this case) in real systems, system comparisons and application characterization using full-system simulation.

CCS CONCEPTS

•Performance, Evaluation, Metrics

KEYWORDS

Benchmark, Performance evaluation.

1 Introduction and Motivation

Nowadays, almost every activity relies on some kind of computing device to work properly. In some of these environments, computation resources run applications of disparate nature concurrently. Cloud computing could be considered the main exponent of this class of workload. The market share of this usage scenario is predominant in many scenarios as it represents a significant fraction of the server processor market [1][2]. In such settings, the responsiveness of an application executing on a shared machine (usually a multi-socket with multi-core processors) might be influenced by other applications running concurrently on the same machine. Therefore, anticipating and/or preventing such cross-effects might have a large economic impact on both the service provider and the final user of the infrastructure.

Performance evaluation, focused on the comparison of alternative architectures/computers, is a fundamental practice for researchers. This task is usually performed making use of a reduced set of applications considered representative of a much broader scenario. Among the multiple suites available [3][4][5][6][7], SPEC CPU (2006 and 2017 [7] versions) is one of the most widely used in the computer architecture community [7]. This suite, developed to be representative of realistic applications, provides a standardized way to measure and compare computationally-intensive performance among different platforms and/or combinations in the software stack and compilers. Unfortunately, the suite is focused on performance metrics that might be inaccurate in the aforementioned usage scenario. The most relevant metric is *SPECrate*, where an instance of the same application is executed in each core of the system under test. Therefore, the *SPECrate* benchmark limits the number of workloads available in the suite to a total of 24. This number might not be sufficiently representative for accurate performance comparison under heterogenous utilization, like the cloud.

In an attempt to improve the limited performance metrics available, previous works have proposed the generation of multiprogrammed benchmarks through the random combination of available applications [8][9][10]. This approach entails a large population of workloads to test. To achieve a feasible performance characterization it is necessary to sample it by selecting a smaller and more representative set. The representativeness of the group selected is usually validated through statistical methods. In most cases these approaches rely on the approximation of the performance metric by a normal distribution [8], applying the Central Limit Theorem [11]. Unfortunately, the number of values required to achieve distribution normality can be extremely high [12], escaling its computational cost. For example, ensuring a successful normality approximation through 1000 workload execution [12] would require nearly 4 execution days to be completed (assuming an average 5-minute execution for each workload). In this paper, we propose an alternative methodology to circumvent this issue. The idea is implemented following the steps described next.

Firstly, we performed exhaustive manual code profiling to identify and label a loop-based ROI in most *SPEC* applications. Next, making use of hardware performance counters, we could perform a detailed per-loop analysis, demonstrating the microarchitectural similarity of the multiple iterations inside ROI loop. This confirmed that the execution of a single iteration of that ROI loop (or a few in some cases) resembles whole execution for many performance

metrics. Limiting execution to that fraction, we could significantly reduce the execution time per workload.

Secondly, we developed a simple lockstep mechanism that synchronizes the execution of the ROI selected for different applications instances across the cores of the system. The source code of every application was annotated including a barrier before the ROI loop, allowing us to pause the application execution in any pre-defined number of iterations of the main loop. This way, mixing different applications on the same processor, we could evaluate the performance effects derived from the competition for shared hardware resources in any of the potential scenarios found in the whole population of workloads.

Table 1. SPEC CPU2017 Summary

Application	Loop	Iterations	Cyc./Iter. (Millions)
500.perlbench	--	--	--
502.gcc	--	--	--
503.bwaves	Yes	80/130/110/120	5730/5771/5382/5574
505.mcf	Yes	33	36820
507.cactuBSSN	Yes	80	10541
508.namd	Yes	65	11372
510.parest	Yes	7	219851
511.povray	Yes	2048	730
519.lbm	Yes	44	266
520.omnetpp	Yes	467	3078
521.wrf	Yes	1440	1512
523.xalancbmk	--	--	--
525.x264	yes	1000/1000/750	162/602/802
526.blender	yes	20	84745
527.cam4	Yes	90	13541
531.deepsjeng	Yes	194	5760
538.imagick	Part	5200	134
541.leela	Yes	970	1830
544.nab	Yes	240	5234
548.exchange2	Yes	54	5121
549.fotonik3d	Yes	1909	883
554.roms	Yes	150	8533
557.xz	Yes	268/344/363	1446/1359/1359

The combination of sampling and synchronization mechanisms allowed us to execute, profile and evaluate a large number of different workloads in a reasonable time. This way, we obtined insights about architectural interactions, by achieving statistically meaningful metrics in a practical way. We exhaustively validated our results against whole-application execution and demonstrated its utility for multiple evaluation experiments: system configuration for optimal performance/fairness, performance comparison beyond *SPECrate* and micro-architectural exploration through simulation tools. The main contributions of this paper are highlighted next:

1. We perform a detailed profiling of *SPEC CPU2017* applications, demonstrating the uniform behavior of main loop iterations for most applications.
2. We present a performance evaluation methodology named *SPECcast*, able to execute and evaluate a large number of different workloads in a reduced time compared to *SPECrate*. We validate our methodology comparing to whole application execution.
3. We propose multiple use cases, demonstrating that *SPECcast* can not only improve performance evaluation accuracy but can also be extended to alternative metrics such as fairness, due to the heterogeneous nature of the workloads, or different micro-architectural parameters, such as cache miss-rate, using performance counters.
4. Finally, we also demonstrate the utility of this methodology in simulation environments, where it can be employed for micro-architectural evaluations.

2 Profiling SPEC

The Region of Interest of an application is the one consuming the largest fraction of execution time and usually devoted to the resolution of the main tasks that the application addresses. In many applications, ROI is composed of a repetitive pattern, in the shape of some kind of iterative control structure such as a for/while statement. Making use of the Linux perf tool [13] to identify the hot execution spots of application code, the first step of this work consisted of finding out whether this loop-based ROI is present in *SPEC CPU2017* applications. A detailed description of the hardware platform employed to perform these experiments is provided in Section 3, under the DESKTOP label. The data collected in Table 1 show the results of our exploration, detailing the presence (or not) of a Loop-based ROI[1], as well as the number of iterations and cycles per iteration when found. As can be seen, we have been able to identify a loop-based ROI in 20 out of 23 applications.

Given the repetitive nature of this kind of control loops, a question that arises from this code analysis is whether a loop iteration of the ROI could be representative of complete application execution. To confirm this assumption, micro-architectural metrics for each iteration should closely mirror whole-execution results and behavior should be similar for every loop iteration. For this reason, to evaluate this hypothesis we must make use of hardware performance counters to collect representative performance metrics for each iteration independently. Considering each metric as a random variable (with a number of samples equal to the number of iterations), we look for average values similar to whole-execution results and a standard deviation that is as small as possible. The similarity analysis makes use of IPC, Branch predictor accuracy and L1D efficiency. The Linux perf tool [13] is employed to collect whole execution metrics. In the case of per-iteration metrics, the source code of every *SPEC* application has been modified in order to include event counting for each ROI iteration making use of the PAPI-C library [14]. Figure 1 shows the results obtained, one graph for each of the metrics evaluated. The two columns in each plot represent whole-execution (TOTAL) and single-iteration (LOOP) values. In the case of iteration results, metrics are collected for every iteration and an average value is calculated. Error bars represent the standard deviation of the set of results obtained.

[1] In every case where a loop-based ROI is detected, this region represents at least 90% of total execution time.

Focusing on IPC results, we observe that most applications show high similarity between LOOP and TOTAL results. 19 out of 26 workloads present a discrepancy of less than 10%. This observation is consistent in other metrics such as L1D Hit Rate, Branch predictor accuracy, IPC, etc. Additionally, the small observed deviation in all of these should be remarked. Error bars are small enough in most cases to consider that micro-architectural metrics remain nearly constant in all iterations.

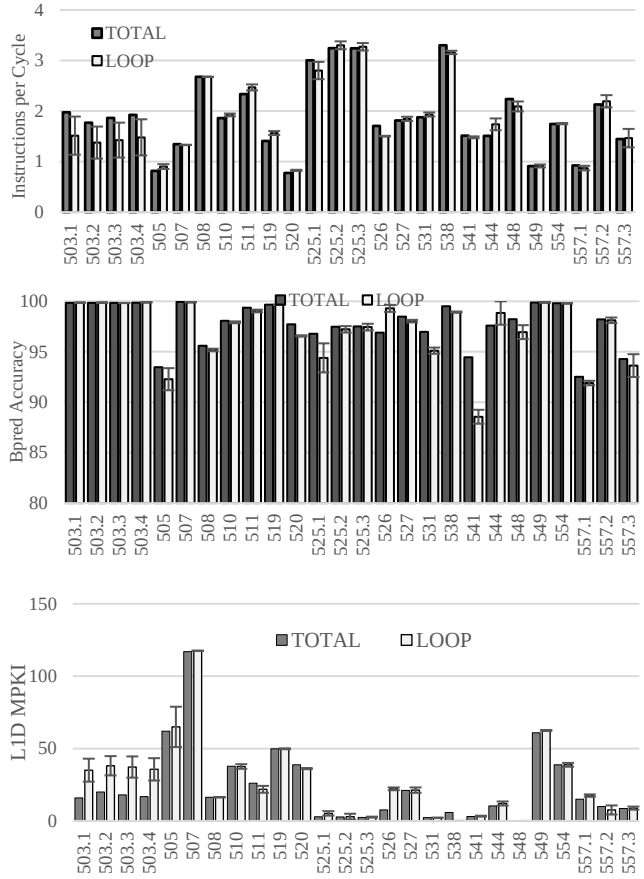

Figure 1. Similarity analysis for different performance metrics: IPC (above), Branch Predictor accuracy (mid) and L1D Miss Rate (below).

Despite the similarity of results in general, it should be noted that in some applications (such as 503) the metric TOTAL deviates from LOOP. In these cases, we have detected two main sources of divergence. First, it must be taken into account that LOOP results are limited to the ROI, while TOTAL results cover the whole application. The metrics collected outside the ROI are therefore a first source of deviation. Second, it can be observed that IPC divergences are combined in some cases with large standard deviation. For those cases, we have detected a variable ROI behavior across different execution phases. For example, Figure 1 shows the per-iteration IPC for 503. As can be seen, two different phases are clearly identified. In these cases, accuracy can be improved by grouping loop phases, characterizing applications as the combination of these phases. If we divide application 503 into two different groups (G1 and G2 in Figure 2), and combine the performance metrics of each group considering their contribution to the total number of loops, we obtain a significant improvement, as shown in Figure 2.

After addressing these accuracy divergences, TOTAL and LOOP results are similar enough for all the applications of the SPEC benchmark evaluated. The results from this section suggest that the mere execution of a small fraction of application ROI resembles the whole execution (concerning micro-architectural behavior). Relying on this feature, the next section introduces the methodology proposed, named *SPECcast*.

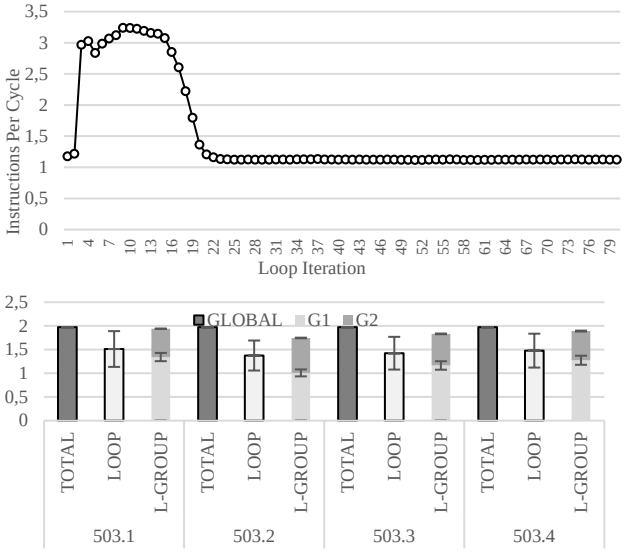

Figure 2. Per-iteration IPC of application 503(above). Accuracy improvement after grouping iterations (below)

3 SPECcast

This methodology tries to implement an alternative evaluation system, still relying on *SPEC* workloads, but making use of a much more realistic configuration and at an affordable computational cost. Through the subsections *SPECcast* is described, validated and analyzed, ensuring its suitability for performance evaluation. For the experiments in this section we use a node configuration representative of two different scenarios: desktop-oriented configuration (DESKTOP) and scale-out server deployments (SERVER). The Desktop configuration is a 4-Core CMP, making use of an Intel i5-7500 chip running at 3.40 GHz with 6MB of cache and a main memory of 16GB. In contrast, the server configuration scales up to 32 Cores, with an Intel Xeon Silver 4216 chip running at 2.10 GHz, with 22MB of cache and a main memory of 110Gb. Both systems use Debian 9 (stretch), with kernel version 4.9.0. All development tools and libraries employed in this work are based on the GNU tool set with the versions of the distribution used. All the code associated with SPEC annotation[2], the methodology proposal

[2] SPEC CPU 2017 is proprietary code. Annotation is released as a software patch that can be applied once the suite has been purchased.

and the experiments of the rest of this document is available through a public software repository[3] (anonymous repository for revision).

3.1 ROI Loop Synchronization

In order to ensure that all the applications in the workload execute at least one iteration of the main loop, *SPECcast* synchronizes all the applications at the beginning of their ROI. *SPECcast* uses a POSIX thread barrier mapped in a shared memory region through a POSIX shared memory object. The barrier and the shared memory object are created by *SPECcast's* main process (or *Master*), which also creates child processes for each application to be launched. Using the annotation of the ROI in the previous section, we add, with minor code modifications, barrier calls at the beginning of the ROI of each application. Parent and children wait at the same barrier until all the *SPEC* apps reach the ROI (usually after at least one execution of the ROI). Then, the barrier is raised.

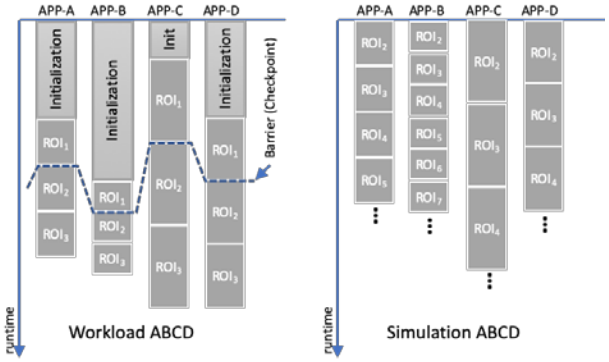

Figure 3. SPECcast Synchronization and execution process.

To ensure minimal system-related noise in the measurement, such as in OS scheduling, we bind each application instance to a different core in the system using the *sched setaffinity* system call. The number of applications and the set used is fully parametrizable. Additionally, a feature is provided to enable the use of PAPI [14] counters to measure the behavior of the SPECs running. The master process attaches a PAPI *EventSet* to each of the application instances and starts the Evaluation when all the processes are ready for execution (i.e. all have reached the barrier). Available performance counters vary with the architecture under evaluation. For this reason, only basic ones such as Total cycles (PAPI_TOT_CYC) and Total Instructions (PAPI_TOT_INS) are included, as they are widely available in x86_64 platforms. The execution either ends when all the applications finish their runs or runs the loop indefinitely until a given time mark is reached.

3.2 Validation

From a statistical point of view, performance can be considered a random variable obeying a certain probability distribution. In order to validate the proposed methodology, this section carries out a similarity test of the performance distributions obtained with *SPECcast* (SCAST label) and with full-application execution described by the rules of execution of *SPEC* (STOTAL label). For STOTAL workloads, each core runs a different application in an "infinite loop", and execution is terminated when every application completes at least one complete execution. For the experiments of this section we run a sufficiently large number of different workloads, each with a random mix of *SPEC* applications. Instructions per Cycle (IPC) will be the performance metric used. The final value is calculated as the average IPC of the applications running on each core of the system under test.

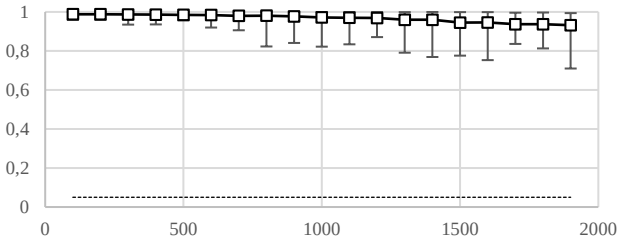

Figure 4. Estimated *p*-value of the Kolmogorov-Smirnov similarity test. Sample size ranging from 20 to 1900 workloads.

Similarity is evaluated making use of the two-sample *Kolmogorov-Smirnov* test [15], a nonparametric test (does not assume data sampled from well-known distributions) that compares the cumulative distributions of two datasets. For a growing sample size n ∈[100, 200, 300, …, 1500] we run the normality test 100 times, generating a different group of workloads each time.

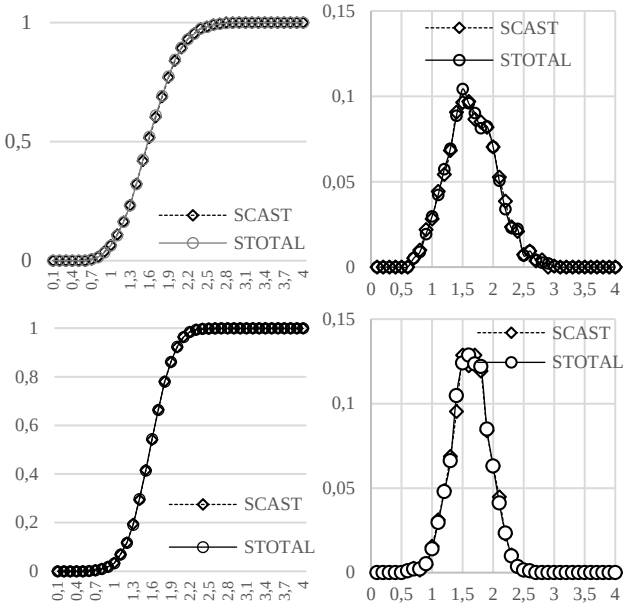

Figure 5. Distribution comparison with a 1500 workload sample. Cumulative frequency (left) and histogram (right). Desktop (above) and Server (below) configurations.

The results in Figure 4 show, for each sample size, the average, maximum and minimum *p*-value obtained from the statistical hypothesis testing. The p-value, or significance probability, is a real

[3] https://github.com/prietop/SPECcast

value between 0 and 1 employed as an indicator for decision-making, A small *p*-value reduces the risk of incorrectly rejecting the NULL Hypothesis (which in this experiment is defined as "both datasets belong to the same distributions"). In contrast, p-values close to 1 tend to confirm this NULL Hypothesis. By convention, pre-defined values employed to reject the NULL Hypothesis are commonly set to 0.05, 0.01, 0.005, or 0.001. As can be seen in Figure 4, for any sample length evaluated, every p-value obtained is >>0,05, meaning that there is an extremely high probability of both sets belonging to the same probability distribution.

This similarity can also be displayed visually in both evaluated systems. For a sample size of 1500 workloads, Figure 5 represents IPC distribution in two different ways, the left graphs represent the cumulative frequency of the IPC distribution, while the right ones show the same information as a frequency histogram. The results above and below correspond to Desktop and Server configurations respectively. As can be seen, both datasets follow a pretty similar curve, it being hard to distinguish between them. These plots confirm the minimal divergences calculated with the Kolmogorov-Smirnov similarity test. Given the large amount of workloads employed for validation, the time spent for this experiment was significant. While SCAST values were collected in less than a day, it took us several days to complete all STOTAL executions.

3.3 Accurate Performance Sampling

The large population size requires the use of sampling in the evaluation procedures, relying on statistical methods to guarantee that the set used is representative of its total population. In many cases, since the applications of each workload are chosen randomly and independently from each other, the Central Limit Theorem is applied. Therefore, performance distribution will be described by a normal distribution. However, in practice it is difficult to determine how large the sample must be to achieve normality. In practice, it could be unfeasibly large from the computational cost stand-point. This section evaluates the effort necessary to achieve normality in different throughput metrics. We have chosen two of the most frequently used metrics [8]: the IPC throughput (IPCT)and the Average Weighted SpeedUp (AWSU). IPCT is defined as the arithmetic mean of the IPC values measured at each core. AWSU is defined as the arithmetic mean of a speedup value calculated as the division between IPC measured at a core and the IPC of the same application running alone on the same machine.

Making use of four sample sizes (20, 50, 200 and 500) we estimate the cumulative frequency graph for 10 different datasets (each dataset is created randomly selecting 20,50,200,500 samples from a 35000-workload population), comparing their distribution to a normal one (with mean and standard deviation values obtained from a sufficiently large population of 35000 workloads). The results in Figure 6 correspond to the IPCT metric, from 20 to 500 samples. As can be clearly seen, for a small number of samples distributions resemble a normal one, but there are significant differences. Increasing the number of samples reduces this discrepancy. Reaching a strong level of similarity has required the utilization of 500 sample datasets, which means that this is the minimum size to guarantee distribution normality. Similarly to the experiments in previous section, while evaluating this number of workloads would extend execution to multiple days, *SPECcast* reduces evaluation time to just a few hours (5-6).

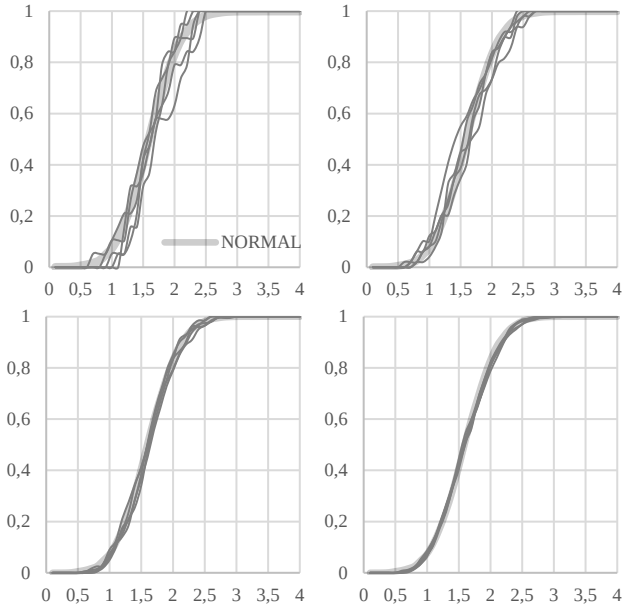

Figure 6. Probability distribution of IPCT metric for different sample sizes: 20 (a), 50 (b), 200 (c) and 500 (d).

AWSU provides a different set of results, as seen in Figure 7. In this case we observe a similar trend to convergence for large samples, but these samples clearly differ from a normal distribution [12]. For the largest sample size, we observe that different datasets converge to similar distribution functions, but this distribution does not fit a normal one. Although not included in the paper, we have extended sample size to 5000 workloads and even for this sample size normality is not obtained.

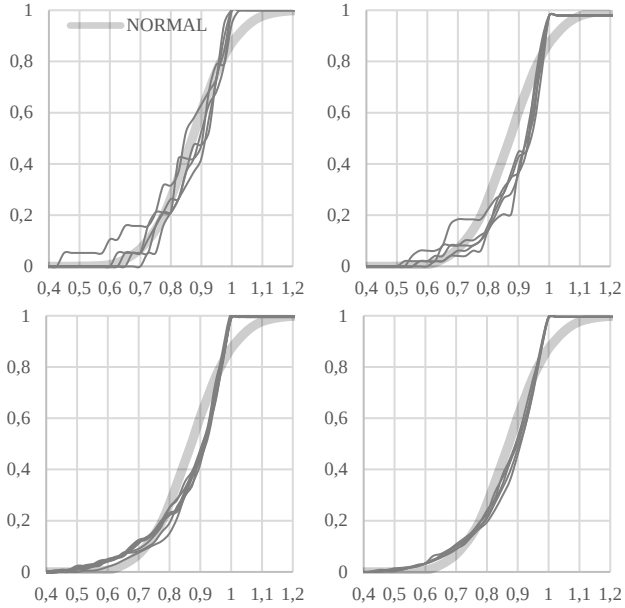

Figure 7. Probability distribution of AWSU metric for different sample sizes: 20 (a), 50 (b), 200 (c) and 500 (d).

SPECcast has enables us to reach a large number of workloads looking for distribution normality. These results show that normality is not always achievable and, in these cases, where it is present, the computational effort required to guarantee it is not negligible. One of the main strengths of *SPECcast* is its reduced execution time, making it feasible to perform statistical evaluations in a reasonable time.

Additionally, the use of the PAPI-C library to collect hardware events extends the metrics available not only to those directly related to performance, but to every micro-architecture feature with an available monitoring event. Thus, aspects such as memory hierarchy performance, branch prediction accuracy or frontend/backend inefficiencies could also be statistically explored using *SPECcast*. Finally, thanks to the utilization of multiprogrammed workloads, we would also be able to extend metrics to those related to quality of service or fairness among applications when using shared resources.

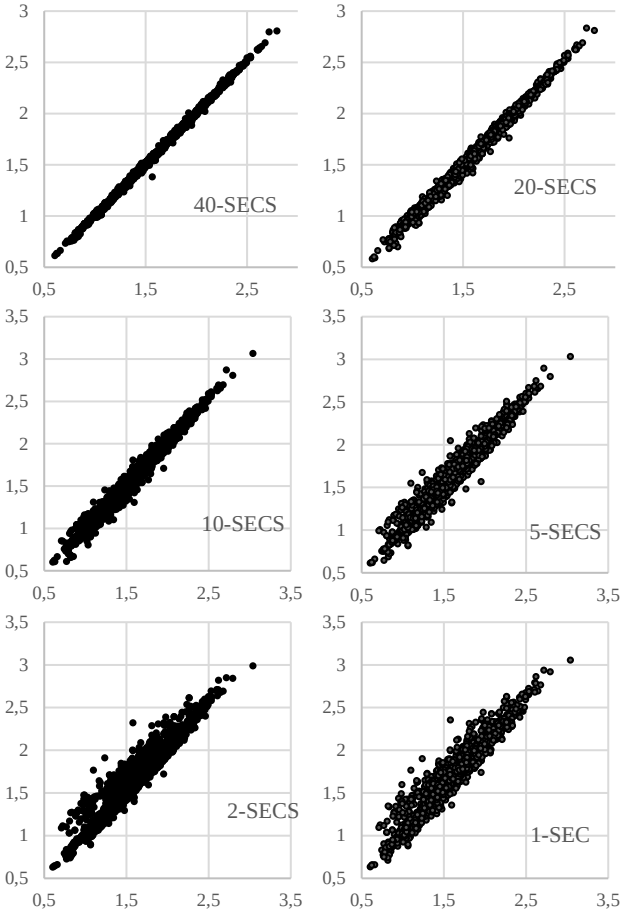

Figure 8. IPC Scatter plot for different execution times of the ROI. Y axis corresponds to IPC obtained from full app execution. X axis corresponds to SPECcast's estimated IPC.

3.4 Speeding Up Evaluation

From the previous section, it is clear that the number of workloads (and time) required for a meaningful statistical evaluation is substantial. In our case, the effort depends mainly on two factors: the time required to reach the synchronization point and the fraction of ROI (number of loop iterations) executed. Previous experiments have been performed for a 60-second ROI execution. However, the size of a single loop iteration (in terms of execution time) is in many cases much less than 60 seconds and this time could therefore be reduced.

Results in Figure 8 analyze, making use of a scatter plot, the degradation of similarity as ROI execution time decreases from 40 to 5 seconds. As expected, as the fraction of ROI executed is reduced, the divergence of results increases. With a small execution time, many applications are not able to complete a loop iteration, harming the assumed similarity. Analyzing the results in depth, running again the two-sample *Kolmogorov-Smirnov* test, we conclude that a 10-second ROI is the lower limit guaranteeing correctness for this specific benchmark suite. This reduction in execution time has a significant influence on the time required to perform a performance evaluation. To illustrate this advantage, the results in Figure 9 show the reduction in evaluation time obtained with *SPECcast*, for the different ROI fractions executed. Values are normalized to the time required to run the whole application. As can be seen, limiting the ROI to 10 seconds we are able to reduce the evaluation time by more than 90%, which becomes relevant when the number of workloads required is close to one thousand, as shown in previous section.

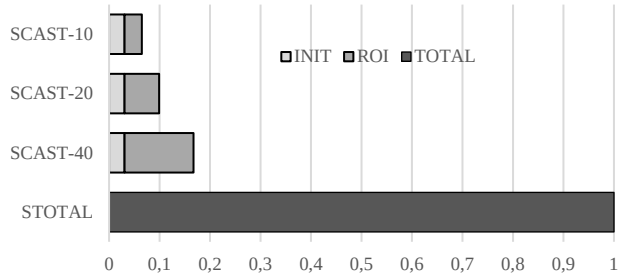

Figure 9. Evaluation time for different ROI execution fractions. Results are normalized to the execution of complete SPEC applications.

After ROI reduction, there is still an unavoidable execution time for every workload, which corresponds to the required time to reach the synchronization point. In this case, a possible way to reduce this time could be to limit the applications included for evaluation purposes. Among all *SPEC* apps, we have detected that a few of them require a much longer time to reach synchronization, as can be seen in Figure 10. If faster evaluations are needed, we could limit the number of applications conforming the workloads, eliminating those with a higher synchronization delay. Of course, these smaller groups reduce the accuracy of the evaluation process, but could be useful as preliminary tests.

From the time-to-barrier results, we can define three application groups: those below 30, 20 and 10 seconds (G1, G2, G3 respectively). As can be seen, with the different groups defined, we are able to reduce the execution time even more. For the most aggressive grouping (SCAST-G3), we reach a 96% time reduction, which means that we can reduce a 2-day evaluation with STOTAL to a 2-hour execution making use of SCAST.

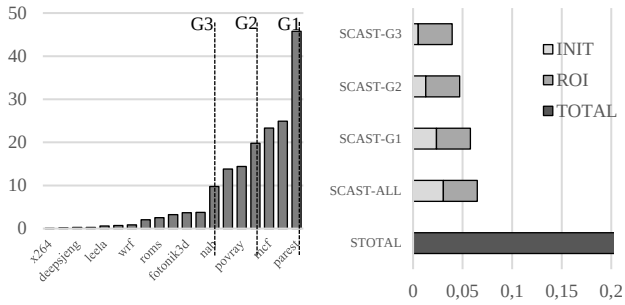

Figure 10. (left) Time to reach the synchronization barrier for each SPEC application. (right) Reduction in execution time derived from discarding slow-sync applications.

4 Use Case 1: Evaluating Prefetch Effect on Performance

As the first usage case, we will compare the system performance benefit with and without hardware prefetching. Recent Intel server processors have two hardware prefetching mechanisms, which can be enabled/disabled through a specific MSR register. These two prefetchers work on the unified L2 cache, detecting a stride (Data Prefetch Logic) or fetching adjacent 64-byte cache lines (L2 Streaming Prefetch). We will run the experiment making use of both the "official" *SPECrate* mode and our methodology. *SPECcast* results are obtained after running 1000 workloads with a 10-second ROI each. The number of *SPECcast* workloads has been chosen to take a similar amount of time to the *SPECrate* evaluation, but in fact it could have been performed in half the time, because 500 workloads is enough to ensure distribution normality, as seen in Section 3.3. The performance comparison is presented in a similar way to [8]. Let PF_{ON} and PF_{OFF} be two random variables corresponding to IPC Throughput (with and without prefetching). According to Section 3.3, both variables have a normal distribution, so we can also define the random variable D as the per-workload throughput difference ($D = PF_{ON} - PF_{OFF}$). As the new random variable D can also be approximated by a normal distribution, the degree of confidence that PF_{ON} is better than PF_{OFF} is equal to the probability that D is positive.

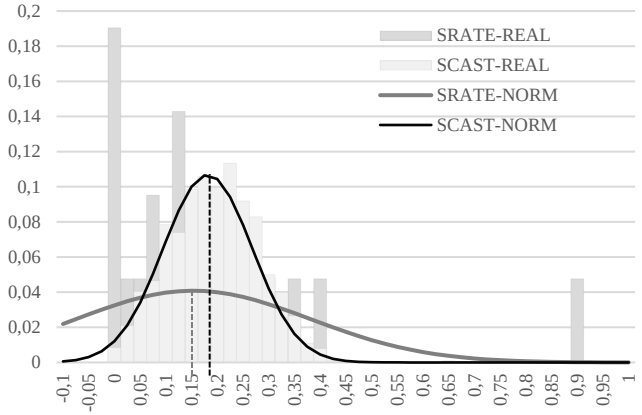

Figure 11. Hardware Prefetching Histograms with *SPECcast* and *SPECrate*.

Figure 11 shows the performance results obtained with both methodologies. The thick lines represent the normal distribution of random variable D for *SPECcast* and *SPECrate* methodologies and the dotted vertical line corresponds to the average value. The background bars represent the real histogram obtained for the workloads evaluated. As can be seen, there is a direct consequence of the reduced number of values provided by *SPECrate*, which is the large variance observed for the normal distribution. In contrast, *SPECcast* is able to provide a performance distribution with a narrower variance. *SPECrate* results indicates that, on average, PF_{ON} provides a 15% performance improvement over PF_{OFF} with a 75% degree of confidence (75% of area below the curve for D>0). Making use of the proposed methodology, we can provide a performance comparison with much more confidence. In the case of *SPECcast*, PF_{ON} provides an 18% performance improvement compared to PF_{OFF} on average, with a 98% degree of confidence (D>0).

As can be seen in this first experiment, our methodology confirms the tendencies observed with *SPECrate*, but also introduces a correction factor derived from heterogeneous workloads (a 20% difference is observed on average values) and improves the comparison, providing a much higher confidence level in this case.

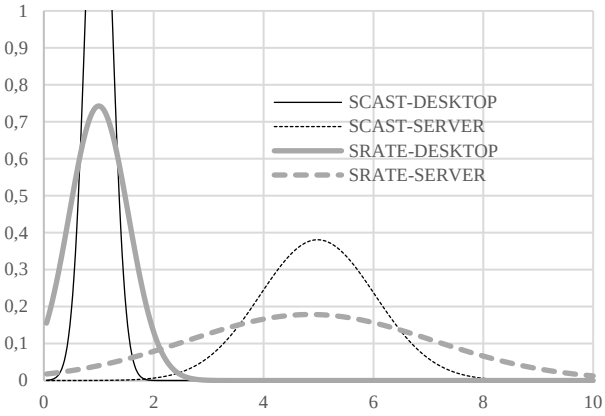

Figure 12. System Performance comparison of Desktop and Server configurations. Probability Density function for each set of results.

5 Use Case 2: System Comparison

Next, we will illustrate how *SPECCast* can enhance the performance insights that *SPECrate* is able to provide. In the experiment we will compare the two systems under test described in Section 3, denoted as Desktop and Server. *SPECrate* metrics are collected through the "official" procedure described in the "Run and Reporting Rules" section. Similarly, we make use of *SPECcast* to generate and run as many workloads as possible in a similar run time, also collecting performance results. In this case, the metric collected for *SPECcast* will be the total number of instructions retired in the interval executed. The results of this experiment are shown in Figure 12, where each curve represents a probability density function of the values measured. Performance values have been normalized to those obtained by *SPECrate* for Desktop

configuration, the mean value for this distribution being equal to 1 in the graph.

Two main conclusions can be extracted from this set of results. First, the similarity of average values (4,98 vs 4,85 for SERVER configuration) confirms that *SPECCast* provides equivalent performance results to official *SPECrate* numbers. This result reinforces the methodology proposed as a complement for *SPECrate* evaluation. Additionally, it can be observed that the significantly larger number of workloads evaluated helps to reduce the standard deviation of performance distributions. Given the statistical formulation usually employed for performance comparison (non-deterministic metric), smaller variance increases the confidence level that SERVER is better than DESKTOP.

6 Use Case 3: Simulation

Reduced computational effort is a clear advantage for performance evaluation of real systems and is even more critical for simulation tools. Previous studies performed significant effort to reduce the time required for detailed hardware simulation [16][17][18][19][20]. Many of these solutions rely on sampling techniques, where only a small fraction of the whole application is executed, while guaranteeing that it resembles the full application. Solutions such as *Simpoints* [16] perform automated code analysis to determine execution phases by similarity . It should be noted that *SPECcast* follows a similar approach, and therefore can be employed in a similar way for simulation. However, it should be noted that simpoints cannot be used to implement the same kind of workloads, since it is not possible to synchronize accurately the execution of application phases. If we limit the detailed simulation to a single iteration of the main loop, our results show that the reduction in execution time can be significant, as can be seen in Figure 13. As can be seen, time saving ranges from one order of magnitude in the worst case to more than 3 orders of magnitude for those applications with short iterations.

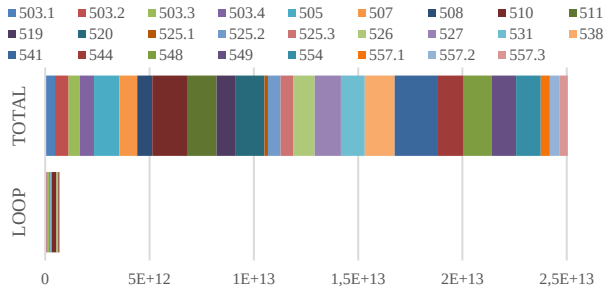

Figure 13. Estimated improvement in computational cost, reducing simulation to a few loop iterationswithin the application's ROI.

Next, we will use the proposal in memory characterization of the suite, where simulation is a better suited framework than real systems, since it enables the alteration of architectural parameters of the system. Previous works, such as [21][22][23][24], have already conduced characterization tasks for *SPECCPU* 20017. In most cases, the characterization has been confined to the use of hardware performance counters in real systems. A few works have employed some kind of simulation framework [23]. Simulation enables much richer profiling, as experiments that are not feasible in real hardware can be performed. One example is the evaluation of the data working set of the applications under test. Results for such an evaluation with simulation tools have been published previously in [23], providing a good opportunity to contrast the results obtained using the *SPECcast* methodology.

For the proposed profiling we make use of the gem5 simulation framework [25]. Gem5 provides detailed CPU and memory system models, as well as supporting many commercial platforms. We make use of virtual machine (VM) based simulation acceleration [26] for checkpointing using full-system simulation mode. During the boot and warmup phases, the VM will run a replica of the simulated machine at near native speed. To achieve a feasible characterization, we only use accurate hardware simulation during the execution of the required iterations of the Region of Interest (ROI) of each application. On reaching the point of interest with VM acceleration, a checkpoint (which includes all architectural state, i.e. processor, memory, network, etc.) is taken. The checkpoint will be loaded subsequently in detailed architectural simulation. Starting from each checkpoint, the memory hierarchy is warmed up for a sufficient number of cycles before starting to collect statistics. In this way we minimize the effect of cold misses and warm-up non-architectural state (i.e. prefetchers, branch-predictors, etc.).

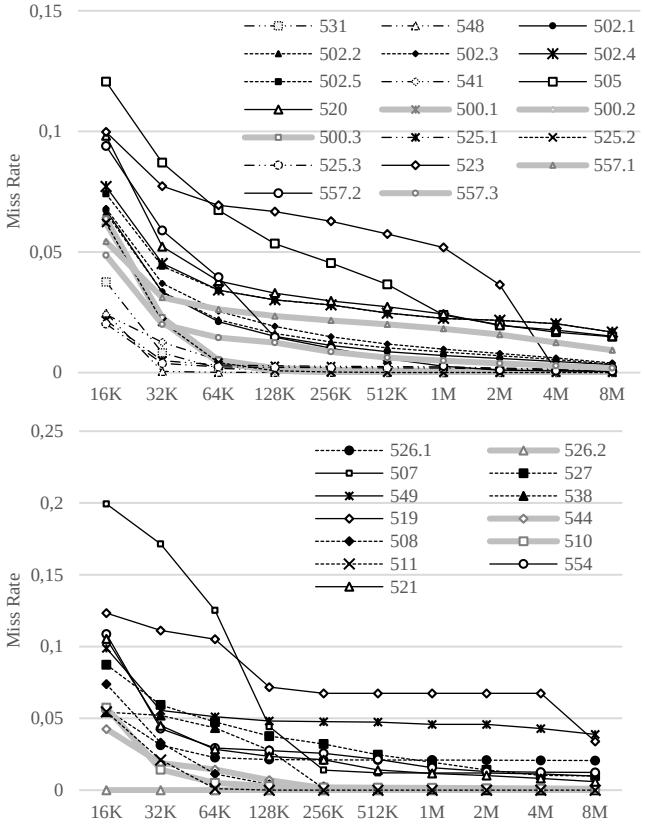

Figure 14. Working set size for SpecInt (above) and SpecFP (below) benchmarks.

For this experiment, we simulate a system with a single core and a single split cache level. For each workload, we conduct a cache sensitivity analysis through the simulation of multiple cache sizes. We modeled a single-level cache hierarchy ranging from 16KB to 8MB. The 16KB configuration is directly mapped, 32KB cache is 2-way associative, 64KB cache is 4-way associative and so on. We use a 64B block size and true LRU replacement policy.

Figure 14 shows a smooth Miss Rate decrease as the cache size grows for most applications. Most workloads suffer from significant miss rates for 16-64Kb cache sizes, Our results confirm those obtained in [23], even the special behavior of some applications. Thus, benchmarks such as *lbm* (519) has a high miss-rate and low cache sensitivity (presents a plain behavior), until 8MB cache size is reached, fitting some of the working set. On the other hand, highly sensitive cache size applications, such as *cactuBSSN* (507), have a continuous miss-rate drop for cache sizes ranging from 16KB to 256KB. *povray* displays similar behavior with a working set size close to 256KB, as miss-rate drops to near zero, the same result seen in [23].

7 Conclusions and Future Work

In this work we have presented a performance evaluation methodology that can overcome the limitations of current alternatives. Making use of profiling and synchronization mechanisms we can generate a huge amount of multiprogrammed workloads running their ROI simultaneously. Additionally, we demonstrate than a small fraction of that ROI is in most cases representative of the whole-program execution.

The methodology has been largely validated, demonstrating its accuracy compared to full execution and its suitability when statistical analysis is required. The methodology provides three major improvements over full SPEC execution: First, it can provide a higher number of measures in the same amount of time, or the same number of measures in a shorter time. Secondly, it allows the measurement of multiple microarchitectural parameters (as many as they are available in the PMU of the system under evaluation).

Finally, it provides hybrid workloads, where different applications run simultaneously on the same system, which enables the exploration of different metrics, such as fairness, which cannot be correctly measured in SPEC rate runs, where all applications running are the same.

We have presented three simple use cases to prove SPECcast's versatility, encouraging future readers to adapt the tools to the huge number of possibilities provided. All the code generated for this work is open access, with the intention of facilitating its utilization by the research community.

REFERENCES

[1] D. Coyle and D. Nguyen, "Cloud Computing, Cross-Border Data Flows and New Challenges for Measurement in Economics," *Natl. Inst. Econ. Rev.*, vol. 249, no. 1, pp. R30–R38, 2019.

[2] E. Jones, "Cloud Market Share – a Look at the Cloud Ecosystem in 2020," *Kinsta Blog*, 2020. [Online]. Available: https://kinsta.com/blog/cloud-market-share/#an-overview-of-the-cloud-computing-market-in-2020.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," *Proc. Int. Conf. Parallel Archit. Compil. Tech.*, 2008.

[4] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," *Natl. Aeronaut. Sp. Adm. (NASA), Tech. Rep. NAS-99-011, Moffett Field, USA*, no. October, 1999.

[5] M. Ferdman and E. Al., "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *ASPLOS'12*, 2012, vol. 40, no. Asplos, pp. 37–48.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, 2010, p. 143.

[7] "SPEC CPU 2017," 2017. .

[8] R. A. Velasquez, P. Michaud, and A. Seznec, "Selecting benchmark combinations for the evaluation of multicore throughput," in *ISPASS 2013 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2013, pp. 173–182.

[9] K. Van Craeynest and L. Eeckhout, "The multi-program performance model: Debunking current practice in multi-core simulation," in *Proceedings - 2011 IEEE International Symposium on Workload Characterization, IISWC - 2011*, 2011, pp. 26–37.

[10] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Representative multiprogram workloads for multithreaded processor simulation," in *Proceedings of the 2007 IEEE International Symposium on Workload Characterization, IISWC*, 2007, pp. 193–203.

[11] L. Le Cam, "The central limit theorem around 1935," *Stat. Sci.*, vol. 1, no. 1, pp. 78–91, 1986.

[12] T. Chen, Q. Guo, O. Temam, Y. Wu, Y. Bao, Z. Xu, and Y. Chen, "Statistical performance comparisons of computers," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1442–1455, 2015.

[13] A. C. de Melo, "The New Linux 'perf' Tools," *Linux Kongress*, 2010.

[14] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing 2009*, 2010, pp. 157–173.

[15] F. J. Massey, "The Kolmogorov-Smirnov Test for Goodness of Fit," *J. Am. Stat. Assoc.*, vol. 46, no. 253, pp. 68–78, 1951.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X) - ASPLOS '02*, 2002, p. 45.

[17] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Conference Proceedings - Annual International Symposium on Computer Architecture, ISCA*, 2003, pp. 84–95.

[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 the Annual International Symposium on Microarchitecture, MICRO*, 2004, pp. 81–92.

[19] 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. 18–30, 2006.

[20] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation," *Parallel Archit. Compil. Tech. - Conf. Proceedings, PACT*, pp. 15–24, 2001.

[21] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?," in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2018, vol. 2018–Febru, pp. 271–282.

[22] A. Limaye and T. Adegbija, "A Workload Characterization of the SPEC CPU2017 Benchmark Suite," in *Proceedings - 2018 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2018*, 2018, pp. 149–158.

[23] S. Singh and M. Awasthi, "Memory centric characterization and analysis of SPec CPU 2017 suite," in *ICPE 2019 - Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 285–292.

[24] A. Navarro-Torres, J. Alastruey-Benedé, P. Ibáñez-Marín, and V. Viñals-Yúfera, "Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP," *PLoS One*, vol. 14, no. 8, 2019.

[25] N. Binkert and E. Al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1, 2011.

[26] S. Bischoff, A. Sandberg, A. Hansson, D. Sunwoo, A. G. Saidi, M. Horsnell, and B. M. Al-Hashimi, "Flexible and High-Speed System-Level Performance Analysis using Hardware-Accelerated Simulation," *Des. Autom. Test Eur.*, vol. 39, no. 2, p. 2012, 2013.