2023 Design, Automation & Test in Europe Conference (DATE 2023) – Best Paper Award Candidate

Perspector: Benchmarking Benchmark Suites

Sandeep Kumar
School of Information Technology
Indian Institute of Technology Delhi
New Delhi, India
Email: sandeep.kumar@cse.iitd.ac.in

Abhisek Panda
Department of Computer Science
Indian Institute of Technology Delhi
New Delhi, India
Email: abhisek.panda@cse.iitd.ac.in

Smruti R. Sarangi
Department of Computer Science
Indian Institute of Technology Delhi
New Delhi, India
Email: srsarangi@cse.iitd.ac.in

***Abstract*—Estimating the quality of a benchmark suite is a non-trivial task. A poorly selected or improperly configured benchmark suite can present a distorted picture of the performance of the evaluated framework. With computing venturing into new domains, the total number of benchmark suites available is increasing by the day. Researchers must evaluate these suites quickly and decisively for their effectiveness.**

We present *Perspector*, a novel tool to quantify the performance of a benchmark suite. Perspector comprises novel metrics to characterize the *quality of a benchmark suite*. It provides a mathematical framework for capturing some qualitative suggestions and observations made in prior work. The metrics are generic and domain-agnostic. Furthermore, our tool can be used to compare the efficacy of one suite vis-à-vis other benchmark suites, systematically and rigorously create a suite of workloads, and appropriately tune them for a target system.

***Index Terms*—benchmarks, hardware counters, Latin hypercube sampling, phase detection**

I. Introduction

Benchmarking is a crucial task that determines how a complex computer system needs to be designed and configured. This information is used for many different purposes such as determining the correctness of a system and optimizing, evaluating, and enhancing its performance. For the better part of the last two decades, benchmark suites such as SPEC[1] and PARSEC [2] proved to be sufficient for the predominantly desktop and cloud-based systems [1, 2]. Designing such benchmark suites is both a science and an art; it often draws on institutional experience adn the intuitions of senior designers.

However, the situation has changed in the past few years. The computing world has seen a rapid proliferation of new devices that are optimized for specific domains of computing such as IoT [3], FaaS [4], edge computing [5], and low-power computing [6]. These domains vary in terms of their requirements and the resources available to them. This has led to a ground-up redesign of the hardware and software in order to extract the maximum performance at a minimal cost. Also, with the advent of the RISC-V ecosystem, hardware design has been democratized. Due to these factors, relying on decade-old benchmark suites [2, 7, 8], which were designed for a completely different purpose, time, and era [9], is perhaps not wise.

The research community has responded by developing new benchmark suites [3, 4, 5, 10, 11, 12, 13] to benchmark these next-generation devices. However, this raises a few important questions, such as do they effectively benchmark the domain for which they are designed? Is there any redundancy among the workloads present in the benchmark suite? Earlier, these questions were answered over a span of several years as the community got more experience using a particular suite. Today, the time-to-market has reduced substantially, and the size of the community for domain-specific processors is very limited.

Furthermore, modern benchmark suites contain many different workloads (e.g., 43 in SPEC'17). However, executing all the workloads is time-consuming and has significant overheads associated with configuring and executing them. Hence, researchers generally execute a subset of these workloads to save time [14]. However, the selection of workloads for the subset is typically *not* motivated by the suitability of the workloads but rather by the ease of configuration and reduction of the total execution time. Using a random set of workloads may lead to misleading conclusions [15, 16, 17, 18, 19].

Hence, there is a need to formally evaluate the quality of a benchmark suite and to quantifiably measure its efficacy, particularly when we are interested in only certain aspects of the overall execution (such as power, TLB misses, or LLC misses). In this work, we present *Perspector*, a novel mechanism to *benchmark such suites*. Essentially, we create a novel set of metrics for measuring the quality of a benchmark suite. Some of these metrics were expressed qualitatively in somewhat nebulous/abstract terms in prior work [2]. However, we are the first to propose a precise mathematical definition. Using this as our fulcrum, our full list of contributions in this paper are as follows:

1) We propose new metrics for assessing the performance of a benchmark suite in terms of its evaluation characteristics.
2) We validate the metrics by evaluating six widely used benchmark suites, including, SPEC'17 [1], PARSEC [2], and Ligra [20].
3) We use the metrics to show how the efficacy of the benchmark suite changes if we specifically focus only on a few select events, such as only cache-related or TLB-related events.
4) We use the metrics to find the right subset of workloads from a benchmark suite that are representative of the whole.

The rest of the paper is organized as follows. We discuss the related work and the motivation for the paper in Section II. This is followed by a discussion on our novel metrics that are used to capture the quality of a benchmark suite in Section III.

978-3-9819263-7-8/DATE23/© 2023 EDAA

TABLE I: Analysis of prior work in this area.

Name	Description	PA?	CA?	Comp?
Workload characterization of SPEC'17 [15]	Statistical techniques to identify redundancy in SPEC 2017	✗	✗	✗
Wait of a Decade [16]	Similarity analysis of SPEC 2017 on different ISAs	✗	✗	✓
Data Analytics Workloads [18]	Similarity between SPEC 2006 and data analytics workloads	✗	✗	✓
Analysis of Redundancy in SPEC 2006 [17]	Statistical techniques to identify redundancy in SPEC 2006	✗	✗	✗
Measuring Program Similarity [19]	Similarity analysis b/w different SPEC variants ans subset generation	✗	✗	✗
Perspector (This work)	Quantifiable approach to compare benchmark suites and subset generation	✓	✓	✓

* PA: Phase analysis, CA: Coverage analysis, Comp: Has a comparison with other suites?

We evaluate Perspector and show its different use cases in Section IV. Finally, we conclude in Section V.

II. Related Work

Prior work in this area [15, 16, 17, 18] has mostly focused on analyzing different versions of the SPEC benchmark suite to find redundancy within it and facilitate the execution of a subset of workloads to save time and effort while maintaining confidence in the results thus produced. Table I shows a brief summary of prior work.

The standard methodology as evinced from prior work is as follows: Theoretically, we define some key parameters that the authors think are crucial. They are mainly the instruction mix, memory operations, TLB operations, cache behavior, and the performance of speculative execution [15, 16, 17, 18]. All the workloads are executed and the relevant data is captured using *performance monitoring units* or PMUs (also known as hardware performance counters). [21]. The dimensions of the captured data are first normalized and then reduced using principal component analysis or PCA [22]. The resulting principal components (PCs) are clustered using hierarchical clustering [23].

Although efficient, this family of approaches has four key drawbacks. ❶ Prior work completely *ignores the execution phases of a workload*. Modern-day workloads show many different phases during their execution cycle. Processing only the final or aggregate values of execution-related hardware counters misses this crucial information. ❷ The clustering process lacks a well-designed metric to quantify the quality of the clusters. This is an important metric that determines the efficacy of the full process. Ideally, a well-balanced benchmark suite *should not form clusters*. The workloads in the suite should be significantly spaced apart from each other and cover the entire execution space. Hence, the hierarchical clustering approach used in prior work to combine workloads needs further scrutiny. ❸ Prior work does not strictly ensure that a benchmark suite holistically stresses all the components of a microarchitecture. A well-balanced benchmark suite should at least explore a few corner cases involving such events so that we are sure that the entire execution space is covered. Ensuring coverage with respect to certain architectural parameters is of vital interest; this aspect has not received its due. ❹ Finally, prior work in this area does not provide an unambiguous method of comparing two benchmark suites. A comparison mechanism is crucial in fields where a researcher has many different benchmark suites at her disposal and would like to select the most suitable one for her experiments. A well-defined and quantifiable metric is required in this case to compare them. The work done [16, 18] contains comparisons between different versions of the SPEC suite and also a brief comparison with some real-world benchmarks. However, the analysis is limited to a mere comparison of the cumulative numbers of a few microarchitectural events and does not capture a holistic view of a benchmark suite's performance.

We tackle all of these issues and present mathematically derived metrics to measure different features of benchmark suites in a quantifiable manner.

III. Benchmark Quality Metrics

In this section, we create a novel set of criteria for evaluating the quality of a benchmark suite. The starting point is inspired from a set of properties listed qualitatively by the authors of the PARSEC suite [2].

1. **Diverse**: The benchmarks in the benchmark suite should be as distinct from each other as possible.
2. **Phase changes:** Modern applications exhibit different phases during their execution. Most synthetic or micro-benchmarks lack this feature. Hence, it is necessary to explicitly consider the phases in real-world benchmarks.
3. **Coverage**: The benchmark suite should cover a very large number of runtime uses cases such that we are sure that all aspects of the system have been thoroughly evaluated.
4. **Spread:** The parameter space should be covered uniformly. We need to avoid clustering in this space, which indicates that two benchmarks have similar behavior.

We devise the following **scores** to mathematically capture the aforementioned criteria.

Notations: Assume that our benchmark suite ($\mathcal{W}$) is a set of n benchmarks, i.e., $\mathcal{W} = \{w_1, w_2, \ldots w_n\}$. During the execution of a benchmark ($w_i \in \mathcal{W}$), we collect m execution-related statistics. We store them in an m-dimensional vector x_i for the benchmark w_i. For a single execution of all the benchmarks in the suite, the matrix $\mathcal{X}$ contains all the individual vectors (stored as row vectors). This description is agnostic to the specific parameters captured in an execution.

A. Diversification: Cluster Score

The benchmarks in a suite should be designed or chosen such that they show different properties as compared to each other in terms of execution statistics. If they are all similar to one another, running different benchmarks will reveal no additional insights. In simple terms, the benchmarks should not be *clustered*.

To capture the "diversity" of a benchmark suite, we define a score called the *ClusterScore*. For this, we first collect the matrix $\mathcal{X}$, normalize it, and create clusters using K-means clustering [24]. To determine the quality of the clusters, we use a well-known score called the Silhouette score [25]. As stated before, the clustering should be as *poor* as possible.

1) A brief description of the Silhouette Score: Let us say we form k clusters. For a point $p \in C_i$, i.e., point p in cluster C_i, we first calculate an *intra-cluster dissimilarity score* for the point p called $\eta(p)$.

$$\eta(p) = \frac{1}{|C_i| - 1} \sum_{\forall p' \in C_i} dis(p, p') \tag{1}$$

where, $dis(p, p')$ is the Euclidean distance between p and p'.

After this, we calculate an *inter-cluster dissimilarity score* for point p, i.e., $\lambda(p)$. We do this for all the clusters except for the cluster containing p, and then select the minimum value.

$$Cost(p, j) = min \frac{1}{|C_j|} \sum_{\forall p' \in C_j, C_j \neq C_i} dis(p, p') \tag{2}$$

Let us define $\lambda(p)$ as the lowest cost for any $j \neq i$.

Finally, we calculate the silhouette score for the point p ($\mathcal{S}(p)$):

$$\mathcal{S}(p) = \begin{cases} \frac{\lambda(p) - \eta(p)}{max\{\lambda(p), \eta(p)\}} & k > 1 \\ 0 & \text{if } k = 1 \end{cases} \tag{3}$$

The silhouette score of cluster C_i is an average of the silhouette scores of all the points within that cluster.

$$\mathcal{S}(C_i) = \frac{1}{|C_i|} \sum_{\forall p \in C_i} \mathcal{S}(p) \tag{4}$$

The silhouette score of a benchmark suite is the average silhouette score for all the clusters. Assuming k clusters:

$$\mathcal{S}(\mathcal{W})_k = \frac{1}{k} \sum_{i=1}^{k} \mathcal{S}(C_i) \tag{5}$$

To calculate the $ClusterScore$, we calculate $\mathcal{S}(\mathcal{W})_k$ for $k = 2$ to $|\mathcal{W}| - 1$, and take the average. Here, k is the number of clusters.

$$\boxed{ClusterScore = \frac{1}{|\mathcal{W}| - 2} \sum_{k=2}^{|\mathcal{W}|-1} \mathcal{S}(\mathcal{W})_k} \tag{6}$$

B. Phase changes: Trend Score

Modern workloads show different phases during execution. Prior work [26] has shown that using hardware counters is an effective method for detecting phase changes during the execution of a workload. We use the same mechanism in our analysis, albeit in a more sophisticated form.

For a given input to a set of workloads $\mathcal{W}$ ($|\mathcal{W}| = n$), let us say $T_z = \{t_1, t_2, t_3 ... t_n\}$ is the set of vectors for a particular PMU (performance monitoring unit) counter $z \in Z$ (e.g. `dTLB hits`) across all the individual workloads. Here, Z is the set of all the PMU counters. Each vector, t_i, is a time series (specific to a given workload).

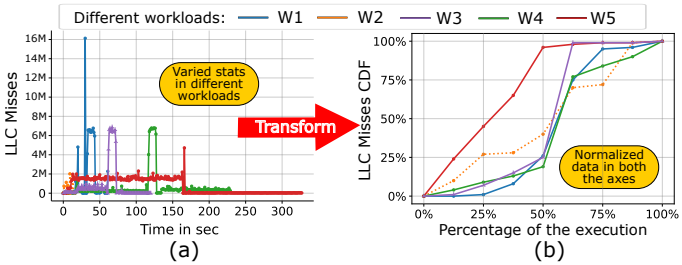


Fig. 1: Normalization of the trend score of LLC misses for five workloads: PageRank, HashJoin, BFS, BTree, and OpenSSL)

To measure the distance between two time series, we use a well-known technique called *Dynamic Time Warping* or DTW [27]. DTW aims to minimize the distance between two time series, which can have different lengths by non-linearly "warping" the time-space to match them.

We calculate the pair-wise DTW distance of all the data points in T_z and report the average for metric z ($TScore_z$).

$$TScore_z = \frac{1}{|\mathcal{W}| * (|\mathcal{W}| - 1)} \sum_{t_m \in T_z} \sum_{t_p \in T_z, m \neq p} DTW(t_m, t_p) \tag{7}$$

We then calculate the average DTW score for all the metrics in Z, which is the final $TrendScore$.

$$\boxed{TrendScore = \frac{1}{|Z|} \sum_{\forall i \in Z} TScore_i} \tag{8}$$

1) Normalization in DTW: Note that if a particular $t_m \in T_z$ has unusually high values (see Figure 1), then the $TScore_z$ score will be dominated by this, and eventually the $TrendScore$. To remedy this, we use the CDF (cumulative distribution function) [28] of a particular metric instead of its absolute values. The CDF normalizes the y-axis. Doing so also bounds the distance between two points to the range [0,100]. However, even now, the x-axis, i.e., the time axis, is still not normalized because different workloads might have taken different times to execute. To fix this, instead of the absolute time, we use the percentiles of the execution time to define the x-axis (see Figure 1). The resultant normalized time series are used to compute the DTW values and, ultimately, the ($TrendScore$).

C. Coverage Score

Depending on the characteristics of the workloads and the input data to them, the relative number of different microarchitectural events can vary significantly. Ideally, workloads in a suite should have a lot of diversity in the number of different microarchitecture events, indicating a good coverage of the parameter space. Here, we define a metric called *CoverageScore* for a benchmark suite, that captures its coverage of the parameter space.

Let us assume we want to compare the coverage of two benchmark suites $\mathcal{W}_1$ and $\mathcal{W}_2$, each containing n workloads. We then execute workloads in $\mathcal{W}_1$ and $\mathcal{W}_2$ for the same amount of time (adjust the inputs accordingly), and then collect the hardware counter matrices $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively, of size $m \times n$.

1) Normalization of the coverage score: The values of the PMU counters can vary from a few thousand to a few billion. In order to compare the scores of different benchmark suites, we need the scores to be within a certain bounded region. Hence, normalization is a crucial step before processing the data. A min-max normalization step brings the data from any range to a predefined range, say [0,1]. However, if we normalize each of the benchmark suites in isolation, we shall lose a crucial piece of information: the relative ranges of the values (two different value ranges, say A:[0 to 10K] and B:[0 to 100K], get normalized to the range [0 to 1]). In order to prevent this, we normalize the PMU counters values jointly.

In order to normalize $\mathcal{X}_1$ and $\mathcal{X}_2$, we first create a matrix $\mathcal{X}$ of size $m \times 2n$ by concatenating $(\mathcal{X}_1|\mathcal{X}_2)$. Then, we calculate two m-dimensional vectors, Q and R, which contain the element-wise maximum and minimum values of the PMU counters across all the workloads.

$$Q_i = \max_{0 \le j < 2n} \mathcal{X}_{i,j} \text{ and } R_i = \min_{0 \le j < 2n} \mathcal{X}_{i,j} \tag{9}$$

Here, $\mathcal{X}_{i,j}$ represents the element at the i^{th} row and j^{th} column in the matrix $\mathcal{X}$. Q_i and R_i represent the i^{th} element of the vector Q and R that contain the maximum and minimum values of the PMU counter m_i, respectively.

The normalization function is as follows:

$$\mathcal{X}\_norm_{i,j} = (\mathcal{X}_{i,j} - R_i)/(Q_i - R_i) \tag{10}$$

Here, $\mathcal{X}_{i,j}$ is the element of the matrix $\mathcal{X}$ and Q_i and R_i are the i^{th} elements of vectors Q and R, respectively. This will bring all the values in the range of [0,1] while preserving the relative difference between them.

2) Scoring: After normalization, we use Principal Component Analysis or PCA [22] to reduce the dimensionality of the data while also preserving its variance. PCA will eliminate all the redundant features (events) from the data. We ensure that 98% of the original variance is preserved.

$$\langle \mathcal{X}_1^T, d_1 \rangle = PCA(\mathcal{X}\_norm_1, \texttt{variance} = 0.98) \tag{11}$$
$$\langle \mathcal{X}_2^T, d_2 \rangle = PCA(\mathcal{X}\_norm_2, \texttt{variance} = 0.98) \tag{12}$$

Here, d_1 and d_2 are the number of PCA components in the transformed data (using PCA), $\mathcal{X}_1^T$ and $\mathcal{X}_2^T$, respectively. We define the *CoverageScore* as the variance present in the transformed data.

$$CoverageScore_{\mathcal{W}_1} = \frac{1}{d_1} \sum_{i=0}^{d_1 - 1} Variance\left(comp_i(\mathcal{X}_1^T)\right) \tag{13}$$

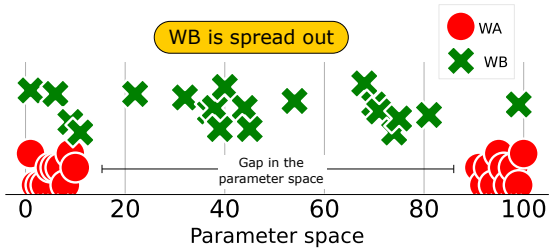


Fig. 2: Difference between coverage and spread. Suite WA has high coverage but a low spread. Suite WB has good coverage and spread.

TABLE II: System configuration

Hardware Settings		
Xeon E-2186G CPU, 3.80 GHz	Disk: 1 TB (HDD)	
CPUs: 1 Socket, 6 Cores, 2 HT		
DRAM: 32 GB	L1: 384 KB, L2: 1536 KB, L3: 12 MB	
System Settings		
Linux kernel: 5.9	ASLR: Off	GCC: 9.3.0
DVFS: fixed frequency (performance)	Transparent Huge Pages: never	

Here, $comp_i$ is the function that returns the contents of the i^{th} PCA component. Similarly, we can calculate the CoverageScore for $\mathcal{W}_2$ also. A high CoverageScore indicates a good coverage of the m-dimensional parameter space (by definition). Prior work [29] has also used a simple variance-based metric to define the coverage of PARSEC and SPLASH-2 benchmark suites.

D. Spread Score

The coverage score gives an estimate of how much variance the workloads have in the parameter space. However, as observed by us and Bienia et al. [29], this analysis is not enough on its own. A few workloads that are different from the mean can cause the variance to inflate (see Figure 2).

To remedy this, we add a metric that measures how uniformly distributed are the workloads of a benchmark suite over the parameter space. Ideally, the workloads should be uniformly distributed and should not leave huge gaps in the parameter space. To this end, we use the KS-test, a well-known test to measure how close a set of points is to a uniform distribution [30]. A KS-score (also known as D-value) in the range of [0,0.5] indicates that the set of points can be weakly approximated as a uniform distribution [30].

$$SpreadScore\mathcal{W}_1 = \frac{1}{n} \sum_{i=0}^{n-1} KS\text{-}Test(\mathcal{X}\_norm_i, U(0,1,m)) \tag{14}$$

Here, n is the number of workloads in $\mathcal{W}_1$, $\mathcal{X}\_norm_i$ is the i^{th} column of the normalized matrix $\mathcal{X}\_norm$ and $U(0,1,m)$ is a set of m randomly drawn points from a uniform distribution between [0,1].

IV. Evaluation

In this section, we discuss the scores assigned to of different benchmark suites by Perspector under different scenarios. The

TABLE III: Description of the different suites and workloads used in Perspector. All the benchmarks are executed with their standard input settings.

Suite	Description
PARSEC [2]	A benchmark suite of parallel workloads to evaluate multi-threading capabilities of a multiprocessor system.
SPEC'17 [1]	A benchmark suite to stress the CPU and the memory subsystem.
Ligra [20]	A lightweight graph processing framework.
LMbench [8]	A set of micro-benchmarks to measure the latency of different system calls.
Nbench [7]	A set of mico-benchmarks to test the speed of integer, floating-point, and memory operations.
SGXGauge [31]	A suite of real-world benchmarks from different domains. *We use the non-SGX versions of these benchmarks.

TABLE IV: Description of the hardware counters.

PMU Counter	Description
`cpu-cycles:`	Total CPU cycles
`branch-instructions:`	Dynamic branch instructions
`branch-misses:`	Branch mispredictions
`dtlb_load_misses. dtlb_store_misses. walk_pending:`	Total #CPU cycles spent in walking the page table for the dTLB load and store misses.
`cycle_activity. stalls_mem_any:`	Total stall cycles
`page-faults:`	Total number of page faults
`dTLB-loads & dTLB-stores:`	Total number of dTLB loads and stores
`dTLB-load-misses & dTLB-store-misses:`	Total number of dTLB load and sore misses
`LLC-loads & LLC-stores:`	Total number of LLC loads and stores
`LLC-load-misses & LLC-store-misses:`	Total number of LLC load and store misses

details of our evaluated system can be seen in Table II. Table III lists the benchmark suites and Table IV lists the PMU events used in this paper.[1] Note that we ensure that the execution times of all the workloads are roughly the same by tweaking the input values.

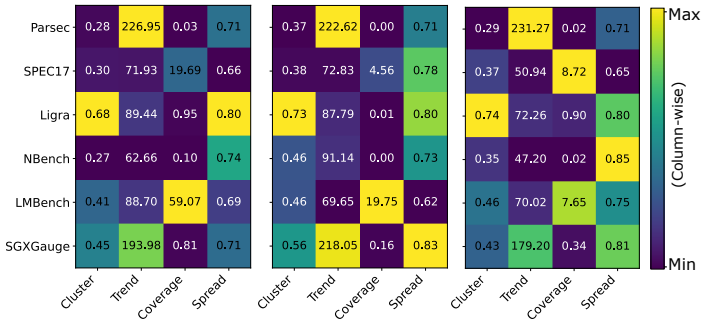
A. Benchmark Suites' Scores

Figure 3a shows the Perspector scores assigned to different benchmark suites using all PMU counters.

❶ *Cluster score* (lower is better): Ligra is assigned the highest score indicating a high degree of clustering. This is primarily because of how Ligra is designed. It is a lightweight graph processing framework for shared memory systems. It mainly consists of two parts, the first component is responsible for loading and decoding the input graph, and the second one is used to implement different algorithms such as BFS and PageRank [20]. As a large portion of the code base is shared, the workloads are expected to behave similarly, justifying the high degree of clustering. The rest of the workloads are fairly spread out, as indicated by their cluster scores (see Figure 4).

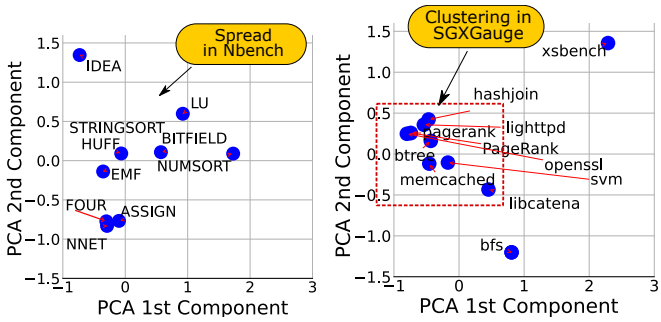
❷ *Trend score (higher is better):* PARSEC and SGXGauge are assigned large trend scores compared to other suites. This is primarily because both of them consist of a diverse set of real-world workloads as opposed to other suites that either just

[1]Capturing more events than the available PMU counters results in a loss of accuracy due to multiplexing by the OS. [21]



(a) All events (b) LLC focused (c) TLB focused

Fig. 3: Benchmark scores for three different settings: a) with all PMU counters b) only using LLC-related PMU counters, and c) only using TLB-related PMU counters.



(a) Nbench (b) SGXGauge

Fig. 4: Clustering in Nbench and SGXGauge.

contain kernels [32], which are susceptible to compiler tuning, or similar workloads (Ligra, LMbench, Nbench). Figure 5 shows the trend of LLC-misses in Nbench and SPEC'17.

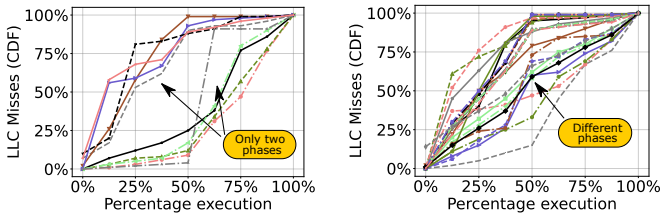
❸ *Coverage score (higher is better):* LMbench has the highest coverage score because its workloads stress different aspects of computing such as the memory bandwidth, IPC bandwidth, and cached I/O bandwidth. Furthermore, it is used for measuring the latency of different OS-related operations, such as reading memory, issuing system calls, and handling signals [8]. This expansive range of testing results in a wide coverage (see Figure 6).

❹ *Spread score (lower is better):* Here, all the suites are assigned a similar score. SPEC'17 performs marginally better due to its workloads' well-spread-out coverage of all the parameter spaces.

B. Focused Scoring

Here, we analyze how the Perspector scores change when we focus only on a few microarchitectural events. This analysis is useful when researchers want to stress test a particular subsystem instead of the complete system.

Figures 3b and 3c show the Perspector score using only LLC-related and TLB-related events, respectively. When using only LLC-events, PARSEC and SPEC'17 have the best cluster score, indicating a wide spread of the workloads. PARSEC and SGXGauge continue to dominate the trend score. LMbench



(a) Nbench (b) SPEC'17

Fig. 5: Trend of LLC misses for Nbench and SPEC'17.

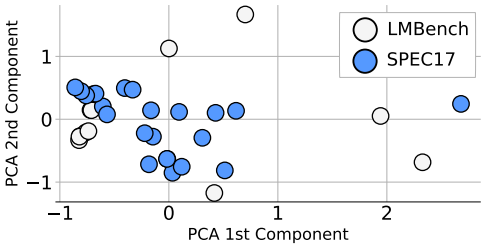


Fig. 6: Figure showing the coverage of LMbench and SPEC'17 using the first two components of PCA [22].

still has the highest coverage score, although it gets reduced by 66%. Similar trends are seen when using only TLB-related events. One key change is in the coverage score. In this setting, SPEC'17 has the highest coverage score. The coverage score of LMbench gets reduced by 88% though.

C. Benchmark Suite Subset Generation

As mentioned before, creating a subset of a benchmark suite is a non-trivial task. As Perspector captures a more holistic view of a program's execution, including phase changes, a subset generator using the proposed m metrics will result in a subset that closely resembles the main benchmark suite. As a proof-of-concept, we devise a novel methodology to generate a subset of a suite based on Latin Hypercube Sampling, or LHS [33]. The LHS method is used to efficiently sample probability distributions in an M-dimensional sample space [33]. In our case, the total number of dimensions is equal to the total number of PMU counters (see Table IV). LHS divides each dimension into fixed regions and then samples one point from each region. The number of regions depends on the number of points that we intend to sample [33]. Using SPEC'17 as a representative suite, we were able to reduce the set of 43 workloads to a subset of 8 workloads using LHS. When compared with the full SPEC suite, the deviation in scores is minimal – just 6.53% (not shown due to space constraints). Similar observations were made by Panda et al. [16].

V. Conclusion

In this work, we introduced four rigorously defined metrics that are broadly in line with qualitative suggestions made in prior work to measure the efficiency of a benchmark suite. Using the metrics, we evaluated the performance of some of the most widely used benchmark suites. We also showed how the benchmark suites fare when we focus on only a particular set of microarchitectural events, how they stand with respect to each other, and how to effectively choose a representative subset of a benchmark suite.

References

[1] SPEC, "CPU 2017," https://www.spec.org/cpu2017/.
[2] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS*, 2009.
[3] A. Shukla, S. Chaturvedi, and Y. L. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *CCPE*, 2017.
[4] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," *Middleware*, 2021.
[5] S. Bäurle and N. Mohan, "Comb: A flexible, application-oriented benchmark for edge computing," *EdgeSys*, 2022.
[6] N. Rajovic, N. Puzovic, L. Vilanova, C. Villavieja, and A. Ramirez, "The low-power architecture approach towards exascale computing," in *ScalA*, 2011.
[7] BYTE, "Nbench," https://www.math.utah.edu/~mayer/linux/bmark.html.
[8] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *ATC*, 1996.
[9] K. M. Dixit, "Overview of the spec benchmarks," 1993.
[10] J. Zbontar, F. Knoll, A. Sriram, M. Muckley, M. Bruno, A. Defazio *et al.*, "fastmri: An open dataset and benchmarks for accelerated mri," 2018.
[11] S. Resch and U. R. Karpuzcu, "Benchmarking quantum computers and the impact of quantum noise," *ACM Computing Surveys (CSUR)*, 2022.
[12] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA'17*.
[13] Y. Wang, S. Liu, X. Wu, and W. Shi, "Cavbench: A benchmark suite for connected and autonomous vehicles," *SEC*, 2018.
[14] S. Singh and M. Awasthi, "Efficacy of statistical sampling on contemporary workloads: The case of spec cpu2017," in *IISWC*, 2019.
[15] A. Limaye and T. Adegbija, "A workload characterization of the spec cpu2017 benchmark suite," *ISPASS*, 2018.
[16] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did spec cpu 2017 broaden the performance horizon?" *HPCA*, 2017.
[17] A. Phansalkar, A. M. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *ISCA*, 2007.
[18] R. Panda and L. K. John, "Data analytics workloads: Characterization and similarity analysis," in *IPCCC*, 2014.
[19] A. Phansalkar, A. M. Joshi, L. Eeckhout, and L. K. John, "Measuring program similarity: Experiments with spec cpu benchmark suites," 2005.
[20] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," ser. PPoPP, 2013.
[21] Denis B., "Pmu counters and profiling basics," https://easyperf.net.
[22] I. Jolliffe, *Principal Component Analysis*, 2011.
[23] "Hierarchical clustering," https://en.wikipedia.org/wiki/Hierarchical_clustering.
[24] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu, "An efficient k-means clustering algorithm: analysis and implementation," *IEEE TPAMI*, 2002.
[25] P. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, 1987.
[26] J. Nomani and J. Szefer, "Predicting program phases and defending against side-channel attacks using hardware performance counters," *HASP*, 2015.
[27] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *KDD Workshop*, 1994.
[28] F. Mosteller, R. Rourke, and G. Thomas, *Probability with Statistical Applications*, 1961.
[29] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors ," in *IISWC*, 2008.
[30] H. Hassani and E. S. Silva, "A kolmogorov-smirnov based test for comparing the predictive accuracy of two sets of forecasts," *Econometrics*.
[31] S. Kumar, A. Panda, and S. R. Sarangi, "Sgxgauge: A comprehensive benchmark suite for intel sgx," in *ISPASS*, 2022.
[32] J. STOKES, "Behind the benchmarks: Spec, gflops, mips et al," https://arstechnica.com/features/1999/04/benchmarking/, 1999.
[33] "Latin hypercube sampling - wikipedia," https://en.wikipedia.org/wiki/Latin_hypercube_sampling.