SHARP: A Distribution-Based Framework for Reproducible Performance Evaluation

Viyom Mittal
Hewlett Packard Labs, USA
University of California, Riverside
viyom.mittal@hpe.com

Pedro Bruel
Hewlett Packard Labs, USA
bruel@hpe.com

Michalis Faloutsos
University of California, Riverside
michalis@cs.ucr.edu

Dejan Milojicic
Hewlett Packard Labs, USA
dejan.milojicic@hpe.com

Eitan Frachtenberg
Hewlett Packard Labs, USA
eitan.frachtenberg@hpe.com

***Abstract*—Performance evaluation studies often produce unreliable or irreproducible results because: (a) measurements have high variability due to multiple system variables and diverse operational conditions, (b) reported results are often focused on point-summary statistics, such as average values. Despite recent efforts, there does not exist a general framework to assess and compare the performance of high-performance systems in a principled and reproducible way.**

This paper addresses this critical gap by introducing SHARP, an open-source framework designed to redefine performance evaluation following a reproducibility-first approach. SHARP enables and facilitates a comprehensive characterization of the performance distribution of an application, while orchestrating experiments efficiently. SHARP addresses these key challenges using (a) robust performance analysis and comparison with *Similarity Metrics*; (b) the automatic determination of a reliable sample size through a diverse set of *Stopping Rules*; and (c) comprehensive recording of experimental conditions and results.

We showcase the need for and advantages of SHARP by evaluating the performance of 20 Rodinia benchmarks on 3 HPC servers with different CPU and GPU configurations. We empirically evaluate SHARP to expose the need for distribution-based statistics, and demonstrate how the stopping rules of SHARP attain reliable performance results while minimizing resource usage up to ∼90% relative to a large fixed number of experiments sufficient enough to establish ground-truth. We see the SHARP framework as a fundamental step towards providing customers and engineers with a reproducible and reliable way to reason and compare the performance of HPC applications and infrastructure.

***Index Terms*—Performance evaluation, Reproducibility, Benchmarking**

I. INTRODUCTION

How can we assess the performance of high-performance Computing (HPC) systems in a reliable and reproducible way? This pivotal question motivates our work. Performance evaluation with HPC systems[1] is challenging due to the inherent variability in systems and the traditional reliance on inadequate point-summary statistics such as mean and median, which fail to capture the full spectrum of system performance. The broader performance evaluation goal can therefore be stated as, how to estimate and compare the efficiency and effectiveness of these systems accurately under noisy or variable conditions. Accurate evaluation of HPC system not only guides where we can invest our resources but also ensures that we can trust our computing systems to drive forward essential discoveries and developments.

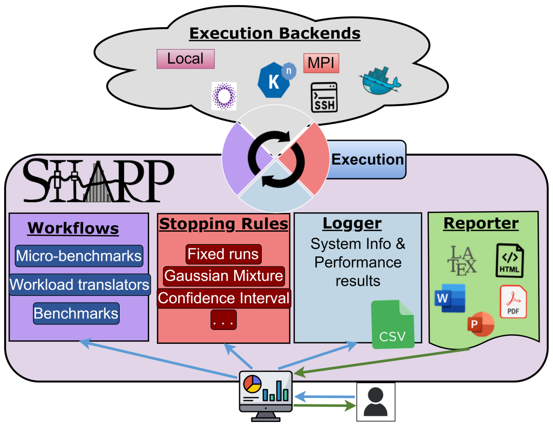
[1]HPC systems: A composition of software and hardware configurations designed for complex computational tasks, also including parallel, distributed, and serverless computing.

Definitions: We use the following terminology in this paper.

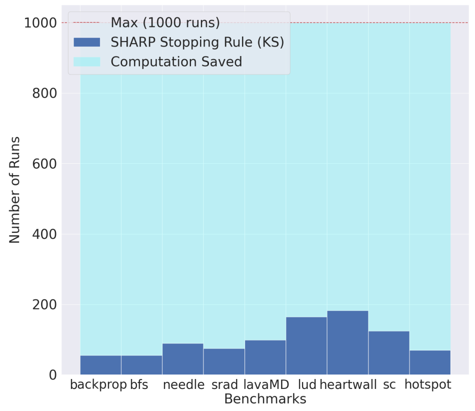
- *Performance Benchmarking*: The process of measuring and comparing the performance of hardware and software systems. In the context of HPC, it involves running a series of standardized tests that assess the speed, efficiency, and reliability of the computing system.
- *Workload*: An application or a benchmark and an input set that is used to test the performance of a computing system. A benchmark, in this context, is a standardized application designed to specifically measure and compare the performance of computing systems.
- *Performance Metrics*: The quantitative measures extracted from benchmarking experiments, such as execution time, throughput, and latency. These metrics are vital for evaluating the efficiency and capability of HPC systems.
- *Reproducibility*: The ability to obtain consistent results using the same workload across comparable computing environments and at different times.

Motivation: Performance evaluation is essential in assessing the efficiency and effectiveness of new systems in both industry and academia. However, achieving accurate and reproducible evaluations is challenging due to various underlying assumptions, methodological choices and hidden sources of noise. Studies have shown that results are often not reproducible and lack statistical rigor, with many relying on simplistic point summaries and no measures of dispersion. Through three illustrative use-cases, this paper demonstrates the need for a reproducibility-focused approach and the utility of the SHARP tool towards this goal.

Problem definition: The overarching problem we address here is the need to evaluate the performance of HPC systems in a reliable and reproducible way. The input to the problem



(a) Performance evaluation cycle with SHARP



(b) Auto-stopping with SHARP (Sec. §V-C)

Fig. 1: SHARP performance evaluation cycle and results

is: (a) a system configuration with its operational parameters, and (b) a workload. The desired output is a reproducible estimate of the performance of the system. For simplicity, we will focus on execution time as the primary performance metric in this work. Clearly, one can consider alternative or more detailed performance metrics such as peak CPU utilization, memory bandwidth, or I/O throughput to capture various aspects of the system's efficiency.

Prior work: There does not seem to be a well-established framework in the HPC community for handling the evaluation of system performance as framed above. At a high level, we group previous works (reviewed in §VII) into two categories: (a) theoretical studies that provide sophisticated statistical methods to evaluate and compare the performance, but unfortunately remain underutilized in practice; (b) practice-oriented efforts that develop performance-evaluation tools, but typically use fixed evaluation methodologies or point-summary-based statistics. To the best of our knowledge, there remains a gap between these theoretical and practical studies. Specifically, we are not aware of any practical tool that deploys sophisticated statistical methodologies for the performance of evaluation of HPC and cloud-based systems.

Contribution: We propose SHARP[2], a systematic and practical framework designed to redefine performance evaluation through a reproducibility-first lens. As shown in Fig. 1a, SHARP provides methods to comprehensively characterize the performance distribution of an application. SHARP handles noise using statistical methods to orchestrate experiment repetitions based on configurable or default parameters. Within this framework, we propose ways to address:

a. How to report and compare performance distributions to make meaningful and reproducible interpretations of the performance data. SHARP embeds detailed descriptions of the **S**ystem **U**nder **T**est, benchmark execution, source code, and runtime parameters in the results, as well as a library of statistical utilities to visualize and interpret the results.

b. When to stop an experiment to balance the reliability of results and the computational resources. SHARP provides several flexible stopping criteria, matching different distribution types, which can be used for stopping the experiment dynamically while ensuring reliability and reproducibility. On top of these diverse criteria, the framework provides a meta-heuristic to characterize the performance distribution in real-time and apply the most appropriate stopping criterion.

As an additional contribution, we implement SHARP as an open-source framework[3] to maximize adoption and impact. SHARP supports various distributed programming models, including Function-as-a-Service (FaaS) without being limited to serverless architectures.

We conduct a detailed performance study using SHARP to showcase how it improves accuracy and reproducibility. Specifically, we assess the performance of 20 Rodinia HPC benchmarks over several high-performance servers with different CPUs and GPU accelerators. Our key findings from this study are:

a. Distributions are better artifacts for performance evaluation than point summaries. We show that distribution-based performance metrics are more robust and accurate than point-summary statistics for performance comparison. An interesting finding is that even while running the same experiment on the same machine for five days, we found that more than half of daily performance distributions were dissimilar across different days. While the distribution-based similarity metric captured this dissimilarity, the point summary-based statistic failed to identify it in cases where the mean value was similar but the other aspects of the distribution like spread and number of modes varied.

b. Our distribution-based auto-stopping algorithm improves both resource efficiency and reproducibility. We show that our dynamic stopping approach manages to outperform prior or fixed-sample approaches. For the Rodinia benchmarks, SHARP uses 89.8% less computation when compared with a fixed sample size large enough to attain "true" performance

[2]SHARP: **S**calable **H**etergogeneous **A**rchitecture for **R**eproducible **P**erformance

[3]https://github.com/HewlettPackard/SHARP

distributions (Fig. 1b). This efficiency is achieved through SHARP's auto-stopping technique, which adjusts the number of runs based on the observed variability, ensuring that only the necessary number of repetitions are performed.

In addition, the performance distribution obtained by SHARP is closer to the "true" performance distribution, compared to the existing approach [1], in which the number of repetitions (100 runs) is either more than necessary or insufficient, varying from application to application.

Finally, we demonstrate the practicality and usage of SHARP by answering three research questions (in Sec. §VI):

Question 1: What key insights are overlooked in point-summary-based performance measurements for HPC benchmarks? We aim to assess the ability of distribution-based metrics to reveal detailed performance patterns, such as multi-modality, which are typically overlooked by simpler statistical summaries like mean or median. With Fig. 4, we find that 70% of the benchmarks exhibit multimodal performance distributions: (a) 40% show two modes, (b) 20% exhibit trimodal distributions, and (c) 10% have more than three modes. SHARP's fine-grained metric collection allows user to customize and collect desired metrics to find and debug the cause of multiple modes as shown in Fig. 7.

Question 2: Which GPU accelerator—Nvidia's A100 or H100 offers better speedup for specific applications? We compared the H100 and A100 GPUs across various Rodinia benchmarks and found that the H100 was consistently faster. However, the speedup varied depending on the application, ranging from 1.2× to 2×. These evaluations using SHARP can assist users in making informed decisions based on application-specific performance gains and hardware costs when selecting the appropriate hardware.

Question 3: How does increased parallelization affect the throughput in a request-response type of workload? We evaluated the performance of the application by sending 2-16 parallel requests and found that while this increased the overall runtime by 39–570%, the execution time per concurrent unit decreased by 30–57% due to parallel processing. This performance analysis using SHARP can help user determine the optimal number of parallel requests the system can handle efficiently while maintaining the required quality of service.

II. Context and Motivating Examples

Performance evaluation is used extensively in the industry and science to assess the efficiency and efficacy of computer systems. However, accurate and sound performance evaluation is still a contentious topic with continuous development [2]. Performance evaluation of computer systems is notoriously difficult to reproduce because of the myriad assumptions, underlying components, and methodological decisions that can affect performance [3], [4]. We discuss these examples below and summarize these examples in the Table I.

Hunold and Carpen-Amarie have shown that many existing MPI benchmarks are neither reproducible nor statistically sound [5]. The majority of these benchmarks just produce point summaries with no measure of dispersion. Sound statistical analysis is necessary to determine whether an observation is repeatable or the result of chance, which is the problem that SHARP attempts to solve. These analyses, however, are not performed universally. Another example of a literature survey reports, after reviewing 112 function-as-a-service (FaaS) performance studies, that most studies do not follow reproducibility principles on cloud experimentation [6].

TABLE I: Key findings and limitations of cited studies

Referenced Studies	Key Findings	Limitations Noted
Hunold and Carpen-Amarie (2016)	MPI benchmarks lack reproducibility and statistical soundness.	Reliance on simplistic point summaries.
Scheuner (2022)	Most Function as a Service (FaaS) studies ignore reproducibility principles.	Poor adherence to reproducibility.
Li et al. (2018)	Evaluated a crowdsourcing framework with small sample sizes.	Limited statistical measures used.
Novo (2018)	Measured IoT architecture performance using averages only.	No uncertainty measures reported.
Heidari et al. (2019)	Introduced Harris Hawks Optimization with variance measures.	Lack of detailed variability descriptions.
Fowers et al. (2018)	Compared AI processor performance on FPGA implementations.	Reported only single summary numbers.
Firestone et al. (2018)	Reported median and percentile performance for SmartNICs on Azure.	Omitted variance details in performance metrics.

To motivate the broad need for a framework like SHARP, we briefly review five arbitrarily selected examples of performance evaluation from different venues and in diverse domains. These examples showcase the pervasiveness and usefulness of performance evaluation on the one hand, as well as its typical summarized reporting on the other hand.

Starting with software examples, Li et al. evaluated a new framework for crowdsourcing by measuring various execution times for 20 sets of image tagging tasks [7]. All 20 times are shown in a scatter plot and their average is reported in the text. This type of performance evaluation, using a small fixed sample size and one or two point summaries is quite commonplace. In a similar example, Novo measured the performance of a new architecture for scalable access management in IoT [8]. The evaluation depicts graphically a single throughput number per concurrency level, representing the average of 5 measurements, again with no uncertainty measures.

Sometimes, the reported performance does add a measure of variability or uncertainty, as is the case for the article introducing Harris Hawks Optimization (HHO) [9]. Although the main tables comparing the performance of HHO to other heuristics do not mention the sample size or even the units of measurement (these are listed in the text), they do include a variance measure for each sample set (standard deviation), but no other description of variability.

On the hardware side, we have two studies that measure the properties of actual hardware. Fowers et al. presented a new processor for real-time AI [10]. The performance evaluation section compared the latency, throughput, and utilization of AI microbenchmarks on three FPGA-based implementations. Only a single number is reported per metric, and it is unclear how it is summarized and from which sample size. A different approach is taken by Firestone et al. when evaluating the performance of custom SmartNICs on the Azure cloud, also using FPGAs [11]. For latency, this paper reports median performance (p50), as well as p99 and p99.9 in 1 million short messages with unspecified variance. Averages are also reported when comparing latency to other solutions, as well as for other metrics, like throughput and end-to-end read time.

All of these examples have three things in common: they are all well-regarded, with hundreds or thousands of citations; they are all fairly recent, so distribution reporting had not been a novel concept at the time; and none of them fully treats performance as a distribution, instead relying on rudimentary dispersion statistics, if that. There is no notion of modality, analysis of long-tail behavior, depiction of uncertainty, or modeling of variability—all of which are paramount for accurate, reproducible, and actionable performance evaluation.

III. SHARP: Methodology

In this section, we discuss strategies for achieving consistent and reliable performance evaluations in high-performance computing systems. Our approach to reproducibility is comprehensive; it ensures the process is robust, the results are reliable, and the interpretations are clear. We achieve this by integrating established theoretical principles with innovative, practical methods. This section is organized into two subsections: **Reproducibility**, where we outline our methods for ensuring dependable results, and **Methodological Innovations in SHARP**, where we introduce new techniques that enhance SHARP's benchmarking capabilities in cloud and HPC environments.

A. Reproducibility

SHARP's *raison d'être* is to facilitate reproducible performance evaluation, so let us define "reproducible" more precisely. One study, for example, defines it as minimizing run-to-run variation in execution performance [12]. While reducing variability is crucial, this definition might be too narrow given the inherent variability in modern systems. Another study suggests that reproducibility in cloud performance measurements requires extensive repetitions and proper statistical treatment [13]. We align with the need for robust statistical approaches but argue against any fixed sample size, which may be insufficient or excessive depending on the context.

Following Peng, reproducibility involves several criteria, including the availability of data, code, documentation, and standardized releases [14]. SHARP adheres to these criteria through open-source practices, containerization, and detailed record-keeping of all experiments. However, these criteria alone do not suffice if performance is considered a static, singular figure.

Expanding on the concept, one study distinguishes three facets of reproducibility: process, results, and interpretation [15]. SHARP addresses these as follows:

- **Process:** Ensuring that experimental setups are well-documented and can be independently replicated, surpassing the basic requirements set by [14].
- **Results:** Moving beyond mean or median performance metrics, SHARP utilizes distribution-based measures, which are more indicative of actual system performance under varied conditions.
- **Interpretation:** By generating more comprehensive reports that include visualizations of data distributions and multiple statistical analyses, SHARP supports consistent interpretations of experimental results.

B. Methodological Innovations in SHARP

SHARP introduces several methodological innovations to address the challenges of performance variability and reproducibility in high-performance computing (HPC) systems:

- **Distribution-based Analysis:** Unlike traditional approaches that focus on point summaries such as the mean or median, SHARP emphasizes the importance of analyzing complete performance distributions. This approach allows for a more detailed understanding of system behavior under various conditions, as recommended by [16].
- **Dynamic Sampling:** SHARP implements adaptive sampling techniques to determine the optimal number of experiments based on the stability and variability of performance data. This method dynamically adjusts the sample size during the experiment, ensuring efficient use of resources while maintaining statistical rigor.
- **Comprehensive Recording and Reporting:** Every experimental setup, run, and result is meticulously recorded and reported in SHARP. This includes detailed metadata on the system configuration and experimental conditions, ensuring that studies are fully reproducible and transparent. This human-readable metadata in turn can be used as input to SHARP to recreate a previous experiment.
- **Non-intrusive orchestration:** Unlike performance analysis tools (e.g., perf) that execute alongside the experiment, SHARP primarily orchestrates the execution of experiments and obtains metrics from the output of the experiments themselves. As a result, SHARP does not interfere with execution and introduces no overhead.

IV. SHARP: Architecture and Implementation

To meet the goals just outlined, SHARP stands on three design pillars: (1) distributions must be captured accurately; (2) distributions must be recorded completely; and (3) distributions must be analyzed and clearly reported with statistical rigor. SHARP itself is therefore not a benchmark, but a framework that runs benchmarks to produce reproducible performance. The framework is flexible enough to handle workflows and workloads on diverse platforms, including local and remote servers, HPC and MPI clusters, serverless (FaaS) environments, and containers.

SHARP runs two classes of executable units (called "functions" in the serverless parlance): Python microbenchmarks suitable for FaaS and black-box programs (user-provided binaries). Microbenchmarks focus on individual aspects of the system's performance (e.g., I/O or MPI synchronization) while programs represent complete benchmarks that focus on application performance (e.g., Linpack). SHARP includes eleven microbenchmark functions, all stateless and atomic, as well as wrapper FaaS functions to run a few open-source benchmarks, such as the Rodinia HPC suite and the GROMACS molecular dynamics benchmarks.

SHARP's architecture, shown in Fig. 2, consists of the following modular Python components:

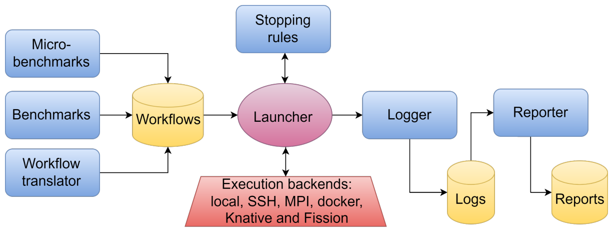


Fig. 2: The SHARP architecture

a) Launcher: The load generator—or launcher—is SHARP's centerpiece. It executes individual functions or programs as prescribed by the workload whilst coordinating the execution backend, the stopping criteria, and the logging. Launcher can be configured for new backends either by deriving from its base class, or in most cases, simply by adding a JSON or YAML configuration file with the required command line invocation. Currently implemented backends include local processes, pre-built docker containers with the benchmark's environment, remote processes (via MPI or SSH), or FaaS invocation (via shell commands or REST APIs that can leverage existing web load generators). The behavior of the launcher is typically controlled via the command line and is highly customizable. Controls include stopping criteria, cold- and warm-start invocations, timeouts, logging and metadata parameters, concurrency, and specific backend options. A particularly useful control is the set of performance metrics to collect, also defined via a simple JSON or YAML interface. This runtime mechanism allows the launcher to collect arbitrary metrics such as latency or power consumption from any function with no code changes.

b) Workflows: Modern workflows often combine different applications or application stages, sometimes with complex dependency relationships. To execute these workflows with their dependency graphs, SHARP uses the time-tested 'make' tool. However, many other workflow formats exist with similar graph information and better syntax. SHARP includes a standalone program to translate workflows from a subset of the popular CNCF's standard Serverless Workflow Specification (in JSON or YAML format) to a valid Makefile (invoking Launcher), which can then be run using 'make'.

c) Stopping rules: One of the key challenges in benchmarking is deciding on the appropriate number of samples (repeated measurements) to collect. Choose too few, and the measurements would be unreliable; choose too many, and precious compute resources would be wasted. SHARP includes eight dynamic stopping rules tailored for specific types of distributions, as well as a novel meta-heuristic to identify the most appropriate stopping rule for the dynamically observed distribution, and another generic rule based on the distribution's self-similarity.

The last two rules require no prior knowledge of the distribution to derive a statistically justifiable sample size. The meta-heuristic requires, however, numerous tunable parameters to classify the appropriate distribution. We tuned these parameters based on a set of 10 synthetic distributions that capture different distributions we observe in real experiments—normal, log-normal, uniform, log-uniform, logistic, bi-modal, multi-modal and autocorrelated sinusoidal distributions—and some distributions that would not really be observed—Cauchy and constant distributions—but that can help test the accuracy of the detection and stopping heuristics. Tuning the detection and stopping heuristics on these distributions is straightforward since most of them have closed-form expressions, and we use large sample sizes (1000 samples) for the ones that do not.

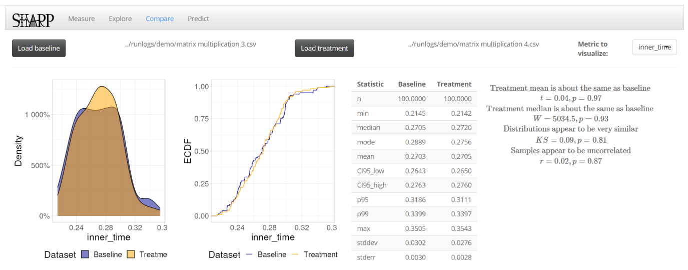
d) Logger: A separate module automates the chore of logging the complete configuration, performance and run data. All metrics and factors are logged in a "tidy data" CSV file to facilitate statistical processing. Logger understands and supports concurrency and parallelism (such as MPI's number of processes or a CPU-based multiprogramming invocation) and records each concurrent instance in its own row. An accompanying markdown description file is automatically written alongside the raw data, describing each field in detail, as well as the metadata required to recreate the System Under Test. Such data includes the description of the hardware, OS, libraries, and software, and even the current git hash of SHARP's own code. This metadata file is both human-readable and machine-readable: SHARP itself can parse it to recreate the same parameters for a reproduction run.

SHARP provides a choice of parameters and metrics to collect and record. The list currently includes the execution time, OS, Memory, CPU and GPU configurations of the system. Adding more metrics and parameters to this list is as simple as adding a YAML file that defines how to collect new metrics or factors from the command line, e.g., using '`/usr/bin/time -v`' to collect the maximum resident size of the program or Linux's 'perf' tool to collect hardware counters.

e) Reporter: The final stage of the benchmark is the statistical processing of the raw data and the human-friendly presentation of results. The independent Reporter module combines RMarkdown [17] scripts with a collection of common statistical utilities for the analysis and reporting of distributions. Executing these scripts on the CSV files that resulted from the workflow execution computes the desired performance metrics, as well as a suite of statistics to quantify uncertainty: means, medians, standard deviations, p-values, confidence intervals, distribution comparisons and visualizations hypothesis testing, and distribution comparisons. The metrics are graphed and uncertainty measures across repetitive function copies are depicted as either confidence/high-density intervals or distribution descriptions. Any change in the underlying platform, software, and hardware can be evaluated by simply rerunning the workflow and reporting.

The resulting reports can be exported to PDF, DOCX, LaTeX, HTML, or PPTX formats. The execution of the markdown files is delegated to a Docker container that freezes all of the statistical and graphing software prerequisites to facilitate the reproduction of the reports. The resulting report includes all of the graphics, statistics, and descriptive narratives.

Graphical user interface On top of these five components and in interaction with all of them, SHARP provides a web-based graphical user interface (GUI). The GUI represents an alternative to running the launcher and reporter from the command line and inspecting the resulting report files, and is particularly suitable for the rapid experimentation stage of the iterative performance-evaluation lifecycle. The GUI currently supports most of SHARP's features, and is planned to also graphically assist the following activities in the future: experiment design, performance prediction, optimization, and automated regression testing. A screenshot of one of the GUI's pages is shown in Fig. 3.



Statistic	Baseline	Treatment
n	100.0000	100.0000
min	0.2145	0.2142
median	0.2705	0.2720
mode	0.2889	0.2756
mean	0.2703	0.2705
CI95_low	0.2643	0.2650
CI95_high	0.2763	0.2760
p95	0.3186	0.3111
p99	0.3399	0.3397
max	0.3505	0.3543
stddev	0.0302	0.0276
stderr	0.0030	0.0028

Treatment mean is about the same as baseline
$t = 0.04, p = 0.97$
Treatment median is about the same as baseline
$W = 5034.5, p = 0.93$
Distributions appear to be very similar
$KS = 0.09, p = 0.81$
Samples appear to be uncorrelated
$r = 0.02, p = 0.87$

Fig. 3: SHARP GUI screenshot of comparison interface

V. Empirical Evaluation

A. Experimental setup

Here, we describe the benchmarks, hardware configurations, and metrics used to evaluate SHARP.

1) Benchmarks: We used 20 diverse benchmarks from the Rodinia HPC suite [18]. Each of the eleven CPU-based and nine GPU-based benchmarks (Table II) were executed 5000 times on three HPC servers, spread across five days with 1000 runs each day.

TABLE II: Benchmark classification and configuration

Benchmark	Parameters
backprop	6553600
backprop-CUDA	955360
bfs	graph1MW_6.txt
bfs-CUDA	graph1MW_6.txt
heartwall	test.avi, 20, 4
heartwall-CUDA	test.avi, 100
hotspot	1024, 1024, 2, 4, temp_1024, power_1024
hotspot-CUDA	1024, 2, 4, temp_512, power_512
leukocyte	5, 4, testfile.avi
srad	1000, 0.5, 502, 458, 4
srad-CUDA	100000, 0.5, 502, 45
needle	20480, 10, 2
needle-CUDA	20480, 10, 2
kmeans	4, kdd_cup
lavaMD	4, 10
lavaMD-CUDA	100
lud	8000
lud-CUDA	1024
sc	10, 20, 256, 65536, 65536, 1000, none, 4
sc-CUDA	10, 20, 256, 65536, 65536, 1000, none, 1

2) Testbed Setup: We used three HPC servers (Table III) to run these experiments. Docker containers were consistently used to ensure uniform libraries across different experiment runs, and precautions were taken to ensure no interfering user processes were running on the system. For each benchmark, execution time was recorded as the performance metric. We cross-checked the performance of the runs within docker and with containers, finding that the container execution overhead was less than 5%.

TABLE III: Hardware configurations

Servers	CPU (cores)	RAM	GPU
Machine 1	AMD EPYC 7443 (48 cores)	256GB	Nvidia A100X 80GB
Machine 2			
Machine 3	Intel(R) Xeon(R) 8468V Platinum (96 cores)	1024GB	Nvidia H100 80GB

We present the histograms of run times of the 5000 executions on Machine 1 in Fig. 4, including boxplots to highlight median and outliers. We choose the histogram bin size as the minimum bin width between the Sturges method and the Freedman-Diaconis rule.

3) Similarity Metrics: To evaluate whether one distribution reproduces another, we employ two statistical metrics to compare multiple runs of identical benchmarks: **Normalized Absolute Mean Difference (NAMD):** provides a relative measure of the average variance between the two sets of data, normalized by their mean values.

$$\text{NAMD} = \frac{1}{2}\left(\frac{1}{\bar{X}}\sum_{i=1}^{n}|X_i - Y_i| + \frac{1}{\bar{Y}}\sum_{i=1}^{n}|X_i - Y_i|\right)$$

Here, X_i and Y_i represent the individual observations in the two sets of experiments, i is the index of summation, n is the total number of observations in each set, $\bar{X}$ and $\bar{Y}$ are the mean values of the sets X and Y respectively.

Implicit Assumption: NAMD assumes that the datasets have the same number of observations and are scale-independent. It is sensitive to the mean of the datasets and does not account for distributional characteristics.

Kolmogorov-Smirnov (KS) Statistic: The KS statistic assesses the distributional similarity of two sets of measurements, with a smaller KS value indicating greater similarity [16].

$$KS = \sup_x |F_1(x) - F_2(x)|$$

The *KS statistic* measures the supremum (the least upper bound) of the absolute differences between the cumulative distribution functions $F_1(x)$ and $F_2(x)$ of two datasets.

Implicit Assumption: KS assumes that the two datasets are approximately independent and identically distributed. This method is non-parametric and does not assume any specific distributional form, making it broadly applicable.

In summary, while NAMD focuses on point summary differences (mean values), the KS statistic considers the entire distribution of performance metrics.

B. Evaluating similarity: point-summary vs distribution based

Aim: The goal of this experiment is to compare the effectiveness of the point-summary metric for performance data against the distribution-based metric, namely NAMD vs. KS.

Experiment: We computed both metrics across 5 days on the 3 machines for the 11 CPU-based benchmarks. For each benchmark-machine, there were 5 performance distributions for each day, which we compare pairwise, resulting in 10

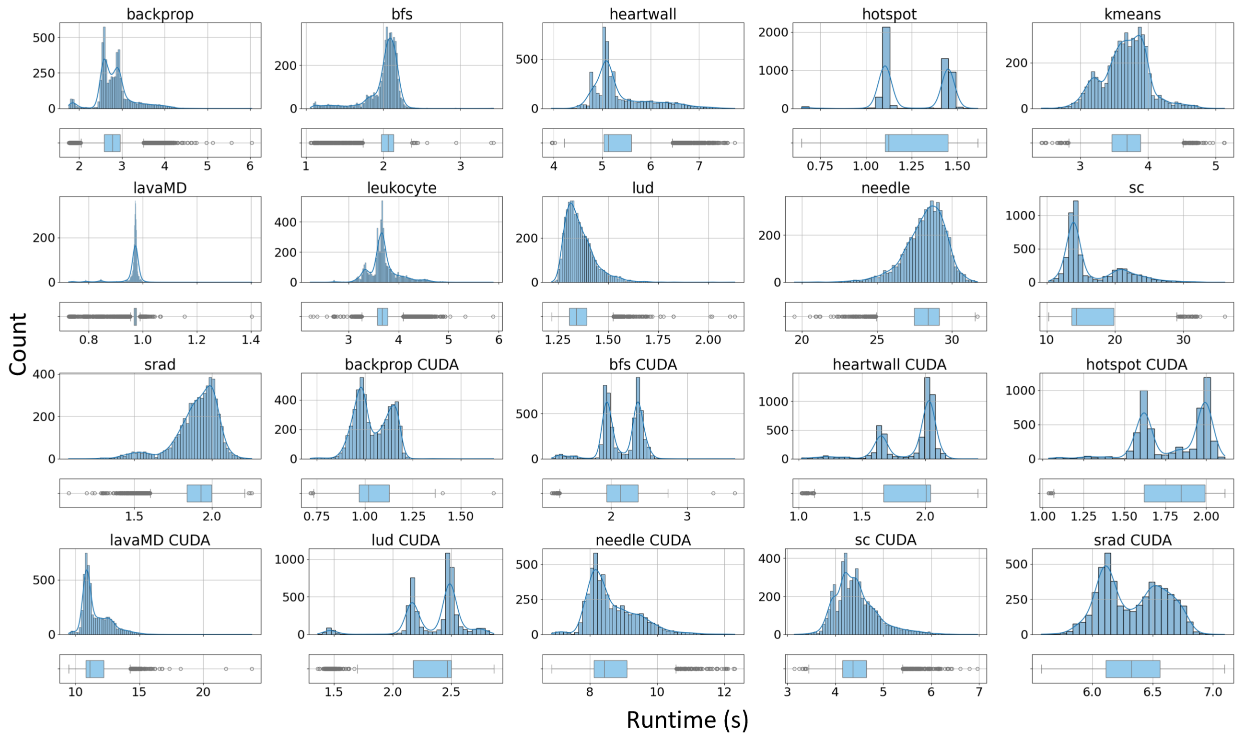


Fig. 4: Distributions and boxplots for 5000 runs on Machine 1

comparisons per benchmark machine and 330 comparisons in total. We utilized a heatmap representation to visually compare the performance similarity metrics across the five day-long runs for each benchmark. Fig. 5b presents one of these heatmaps for the Hotspot Rodinia benchmark as an example of the differences between the metrics. We also compare all NAMD and KS statistics in the scatter plot in Fig. 5a.

Result: We observe that many points exhibit high KS distance but low NAMD values, which indicates cases where the mean values of the distributions are similar, but other aspects, like distribution spread or number of modes, differ significantly. These important differences can be overlooked by NAMD. Therefore, the KS statistic is preferable for comparing two distributions as it captures more than just the average values. We highlight one case between the 3^{rd} and 5^{th} day runs of the Hotspot Rodinia benchmark on Machine 2 in Fig 5b. The NAMD value was zero, suggesting a high similarity, while the higher KS value (0.21) suggests a more significant difference in distributions. Fig. 5c shows that the distributions of the two runs are quite different indeed, despite the similarity in their means. While the 3^{rd} day run has three modes of execution time, there are only two modes in the 5^{th} day run, a distinction that is often missed when using point-summary comparisons.

Takeaway 1*. Point-summary metrics fail to compare effectively for distribution features such as modes and tail, and are therefore inferior to distribution-based similarity metrics.*

C. Evaluating stopping criteria: summaries vs. distributions

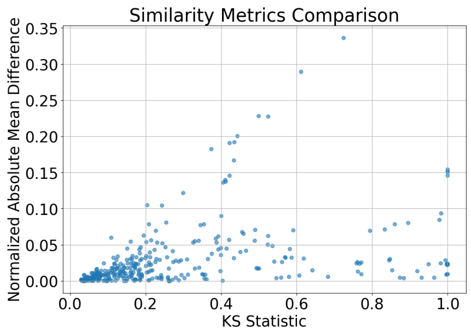
Aim: This experiment aims to evaluate the efficacy of different stopping criteria in benchmark experiments for heterogeneous environments with different hardware. These criteria help in stopping the experiment automatically at a point when the sample size is just large enough to estimate performance robustly, thereby saving resources. Here, we compare the traditional fixed-number-based stopping criterion—as used in the SeBS framework [1] and the studies in Sec. §II—against dynamic stopping rules based on confidence interval (CI) and Kolmogorov-Smirnov (KS).

Dataset: We gathered the performance distributions of the GPU-based Rodinia benchmarks on the Knative serverless environment with Machine 1 and 3 as worker nodes. During each run, we sent two parallel requests to Knative which were divided and executed on A100 (Machine 1) and H100 (Machine 3). From our previous comparisons for NAMD and KS statistics, we found that an upper limit of 1000 runs is adequate to reproduce the performance distributions.

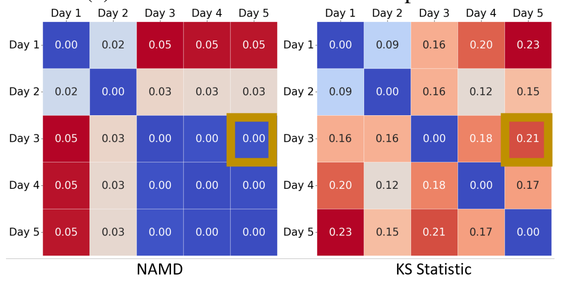
Experiment: We reran the benchmarks with Knative on Machine 1 and 3 with these three stopping rules:

- **Fixed:** The fixed stopping rule stops the experiment after a fixed number of 100 runs, as recommended in the SeBS framework [1].
- **Confidence Interval:** The CI heuristic stops when the 95% right-tailed confidence interval of all run-time measurements is smaller than a threshold proportion of mean (we used two CI thresholds, shown in Table IV).
- **Kolmogorov-Smirnov:** The KS-based stopping rule calculates the KS between the 1^{st} and 2^{nd} half of the runs and stops when it drops below the given threshold.

In Fig. 6, we show the amount of computation performed (or conversely, saved) when using different stopping rules. The second and third panels show the NAMD and KS differences of the partial samples to the complete 1000-run dataset.



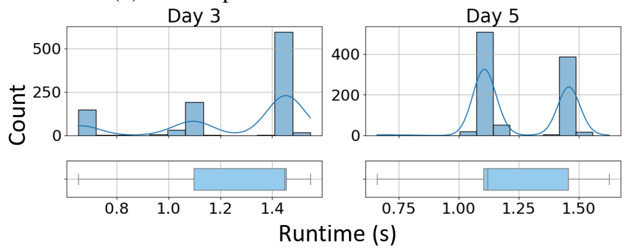
(a) NAMD and KS statistic comparison



NAMD	Day 1	Day 2	Day 3	Day 4	Day 5
Day 1	0.00	0.02	0.05	0.05	0.05
Day 2	0.02	0.00	0.03	0.03	0.03
Day 3	0.05	0.03	0.00	0.00	0.00
Day 4	0.05	0.03	0.00	0.00	0.00
Day 5	0.05	0.03	0.00	0.00	0.00

KS Statistic	Day 1	Day 2	Day 3	Day 4	Day 5
Day 1	0.00	0.09	0.16	0.20	0.23
Day 2	0.09	0.00	0.16	0.12	0.15
Day 3	0.16	0.16	0.00	0.18	0.21
Day 4	0.20	0.12	0.18	0.00	0.17
Day 5	0.23	0.15	0.21	0.17	0.00

(b) Heatmaps of NAMD and KS statistic



(c) Distribution of 3rd and 5th day runs

Fig. 5: Heatmaps and distributions for hotspot on Machine 2

TABLE IV: Thresholds for stopping rules

Stopping Rule	**Stopping Condition**	**Threshold**
Fixed	100 runs	None
Confidence Interval	$CI < T$	$T_1 = 0.05$ $T_2 = 0.01$
Kolmogorov-Smirnov Rule	$KS < T$	$T = 0.1$

From this experiment, we can conclude the following:

- The fixed stopping rule, while straightforward, does not adapt to the variance seen in performance distributions, either stopping too early or too late (for the full 1000).
- CI-based stopping rules, particularly using the tighter threshold T2, tend to continue the experiment longer than necessary to achieve a minute improvement in precision.
- The KS-based stopping rule exhibits a balanced approach, significantly reducing the number of runs while maintaining result reproducibility (up to 89.8% less compared to the default 1000, hence the savings reported in Fig. 1b). The implication is that the KS rule is effective in identifying when additional runs cease to provide new information about the performance distribution.

***Takeaway 2**. KS-based stopping rule reduced the computation by 89.8% for this suite while maintaining a low KS divergence value of 0.104 when compared with the ground truth.*

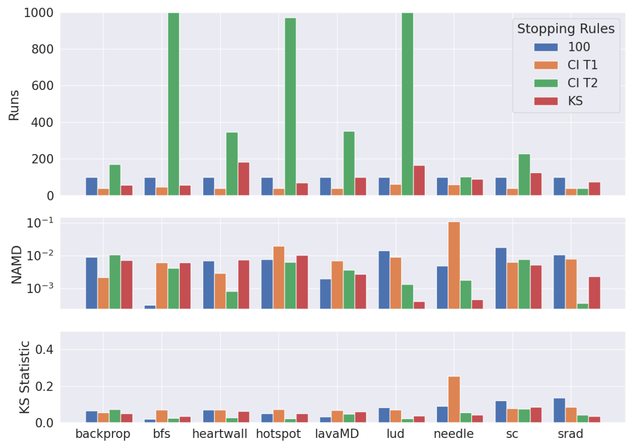


Fig. 6: Comparison of stopping rules on Machine 3

VI. Use Cases for Sharp

We next discuss additional use cases for the distribution-focused approach using Sharp.

A. *Use case 1: Fine-Grained Application Analysis*

We demonstrate Sharp's capability to collect and summarize fine-grained performance for detailed application analysis by analyzing the leukocyte tracking application. The user-configurable metrics are captured concurrently and include:

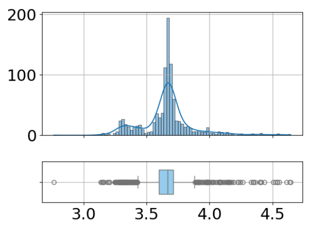
- ***Execution Time*****:** Overall time taken to run the entire leukocyte tracking application, involving both the detection and tracking phases (explained next).
- ***Detection Time*****:** Time spent in the detection phase, which includes the Gradient Inverse Coefficient of Variation (GICOV) computation to enhance edge detection and the subsequent dilation to smooth the edges.
- ***Tracking Time*****:** Time spent in the tracking phase, which involves computing the Morphological Gradient Vector Flow (MGVF) and the evolution of the snake algorithm (active contour model) to fit the edges of the leukocytes.

Insights for the user: From the distributions in Fig. 7, we see that the dual modes in the overall execution time were introduced in the tracking phase. This insight provides users with the knowledge of the cause of the introduction of two modes so that they can focus on optimizing the tracking phase. Overall, users can customize Sharp to log desired metrics from the application for fine-grained performance analysis.

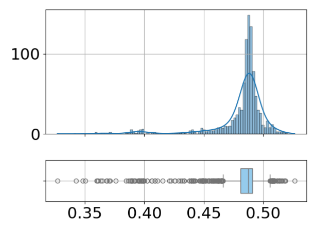
B. *Use case 2: Comparing GPU accelerators*

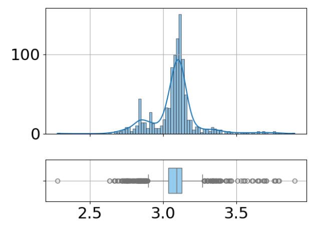
We compare the performance distributions for Rodinia CUDA-based benchmarks on Machine 1 with A100 and Machine 3 with H100 GPUs. The H100's distributions exhibit up to 2× faster performance and with more modes than A100 distributions, which may imply diverse performance states within a single benchmark.

Insights for the user: These characteristics inform users about expected performance differences. For example, in the *bfs* CUDA benchmark (shown in Fig. 8), the H100 shows a potential speedup of up to 2× compared to the A100 with higher modes to the left indicating faster performance in most



(a) Distribution of Execution Time

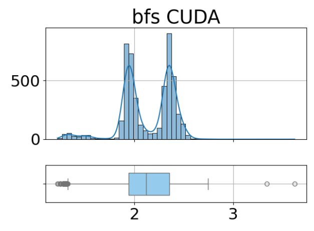


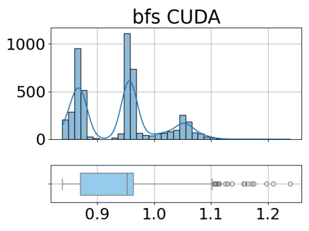


(b) Distribution of Detection Time (c) Distribution of Tracking Time

Fig. 7: Performance breakdown for fine-grained analysis of the leukocyte tracking application

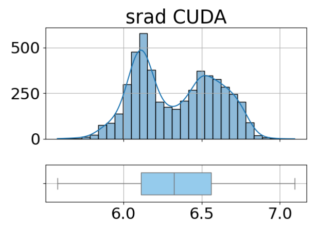
cases. For *srad* (shown in Fig. 9), the H100 provides a more modest speedup of $\approx 1.2\times$. These evaluations can be performed with ease using SHARP and help users make informed decisions based on the hardware cost and the specific performance gains which may vary significantly for different workloads.

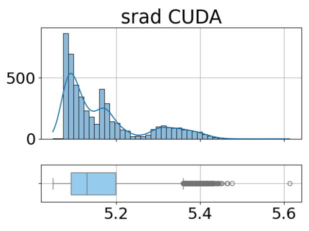




(a) A100 (b) H100

Fig. 8: *bfs* performance comparison for A100 and H100





(a) A100 (b) H100

Fig. 9: srad performance comparison for A100 and H100

C. Use case 3: Evaluating the effect of parallelism

We study the impact of concurrency or parallelism on the Stream cluster (*sc*) task on Machine 3. We ran the workload with increasing levels of concurrency and recorded the average execution time across the runs. The results are summarized in Table V. We also report execution time per concurrency unit, which is the average execution time divided by the concurrency level. This metric helps in understanding how well the system handles increasing levels of parallel tasks.

TABLE V: Effect of concurrency on application *sc*

Concurrency	Avg. execution time (seconds)	Execution time per Concurrency Unit (seconds)
1	3.46	3.46
2	4.80	2.40
4	6.87	1.72
8	11.90	1.49
16	23.14	1.45

Insights for the user: As the level of concurrency increases, the average execution time increases. The execution time per concurrency unit decreases, indicating that the system scales well with increased concurrency. Users can leverage concurrency to handle larger task loads more efficiently. SHARP helps users make informed choices for concurrency to efficiently provision system resources within a given quality-of-service envelope.

VII. RELATED WORK

As mentioned in Sec. §II, methodological pitfalls still affect much of the performance evaluation practice, even in peer-reviewed scholarly publications. There is an extensive body of work on this topic, and we can only survey a few particularly relevant studies in the scope of this paper. These studies range the gamut from the theoretical (describing principles for reproducible performance evaluation, improving accuracy, or addressing statistical pitfalls) to the practical (predicting or reducing performance variability and describing tools and implementations to achieve all of the evaluation's goals). We briefly survey some of these works in the same order of ideas.

A recent paper by Papadopoulos *et al.* lays out eight principles for reproducible cloud-performance evaluation, including experiment repetition with adequate confidence measures, a full description of the experimental setup and metadata, probabilistic result description, and statistical evaluation of significance [2]. These principles are adhered to closely in the SHARP benchmarking framework. Similarly, SHARP follows many of the practices for reproducible performance evaluation in parallel computers proposed by Hoefler and Belli [19]. This paper covers common pitfalls of experimental design, experimental data analysis, comparing statistics, measurement, and reporting. Examples include neglecting confidence intervals, assuming collected data is normally distributed, handling outliers incorrectly, comparing central-tendency point summaries in complex distributions, and incomplete setup documentation and measurement reporting.

To compare the performance of two systems, Hunold *et al.* suggest looking at distributions of averages (mean or median) of repeated experiments using hypothesis testing (t-test or Wilcoxon, respectively) [5]. However, these methods may still produce misleading results, as shown in Sec. §V. Alternatively, De Olivera *et al.* suggests that when comparing two performance distributions and the effect of a variable, quantile regression is more reliable than ANOVA [20]. SHARP takes a step further and compares the similarity of the complete distributions themselves. It also fully records them in CSV files so that any additional tests and analyses like quantile regression or hypothesis testing can be carried out with ease.

There are other examples of benchmarking frameworks that incorporate subsets of the principles adopted by SHARP. Popper [21] is the name given to an experimentation methodology that increases reproducibility by combining the following elements: version control; package management; orchestration and environment capture; infrastructure automation; dataset management; data analysis and visualization; performance monitoring; continuous integration; and automated performance regression testing. Beyer *et al.* discuss challenges in reproducible benchmarking of complex programs on a modern node and suggest some Linux tools (and a framework) to control and limit some of the sources of variability [22].

On the cloud front, some frameworks specifically address the variability introduced by interference and networking noise [23], [24]. The Duet procedure for cloud benchmarking is based on the assumption that performance fluctuations due to interference tend to impact similar tenants equally, and attempts to maximize the likelihood of such equal impact by executing the measured artifacts in parallel [25]. Similarly, Kuhlenkamp *et al.* presented an experiment design and toolkit to measure the elasticity of FaaS services and evaluated it on AWS, Azure, GCP, and IBM [26].

With the recent emergence of serverless computing, a number of benchmarks and microbenchmarks have been proposed as well, albeit for homogeneous architectures [1], [27], [28], [29], [30], [31]. The high level of abstraction and the opaqueness of the operational side make the reproducible evaluation of serverless platforms particularly challenging [32]. For example, Eismann *et al.* show that Microservices can exhibit high response-time variability, even across repeated runs of the same experiment [33]. Regression testing of the variability can be accomplished with enough repetitions and using the Mann-Whitney U test. Another framework that is more application-centric, BeFaaS, was presented by Grambow *et al.* and evaluated on AWS, GCP, and Azure [34]. Similarly, Barcelona-Pons and Garcia-Lopez presented a framework called SeBS that emphasizes the performance of parallel applications, evaluated on AWS, Azure, GCP, and IBM [35]. The SeBS study is also notable in that it specifically examined the variability of various serverless benchmarks on commercial platforms and not just its mean performance [1], and we use it as a reference point for the empirical evaluation. Another popular framework for microservices (not necessarily serverless) called DeathStarBench includes diverse types of CPU applications [36], and a suite of GPU applications by Danalis *et al.* was implemented in both OpenCL and CUDA [37].

To reduce performance variability in such benchmarks, Mariani *et al.* use hardware-independent counters (based on the PISA tool from LLVM's IR) to build a model to predict performance on the cloud. Their work uses *design of experiments* to minimize the effect of noise on the predictions. It predicts median performance, so it does not address variability beyond trying to reduce it.

As a complementary approach, Patki *et al.* treat variability (or the lack of reproducibility) as something to be traded off for performance. Their study defines a desirability metric for run times as $e^{-mean \times variance}$ and attempts to maximize it, or in other words, to minimize variability and run time concurrently. The method used borrows from graph signal analysis to identify the parameters that have a strong influence on performance or variability [12]. On the other hand, a large-scale study of cloud performance data shows that variability is inescapable, even when careful experimental methodology is used. The reviewed performance studies rarely report variability and use only a few repetitions per experiment. This survey's recommendation is therefore to use "sufficient repetitions and sound statistical analyses" [13].

VIII. CONCLUSION

The performance evaluation of high-performance systems needs to grow more robust to enable reliable and informed decisions both in research and industry.

The key contributions of our work are twofold: (a) we quantify and explore the variable, unreliable and intertwined world of performance evaluation in HPC systems, and (b) we propose SHARP, an open-source benchmarking framework that addresses these issues by following a reproducibility-first approach. Going beyond the widely-used point summaries of performance, SHARP focuses on the *performance distribution* as the key objective for reproducible assessment and comparisons. Enabling this idea in practice involves several significant challenges: (a) comparing similarities of performance distributions; (b) identifying optimal sample size for reliable performance results; and (c) maintaining similar system and experiment states across different runs.

Within SHARP, we develop effective ways to address these key challenges by providing: (a) a distribution-centric performance analysis approach based on *Similarity Metrics*; (b) an automatic determination of the appropriate sample size through a diverse set of *Stopping Rules*; and (c) comprehensive recording of experimental conditions and results to further support reproducibility. We evaluate our approach using 20 Rodinia benchmarks on 3 HPC servers with different CPU and GPU configurations. We demonstrate the need for distribution-based statistics by showcasing their advantages over point summaries. We also show the effectiveness of the stopping rules of SHARP in attaining reliable performance results while minimizing resource usage up to $\sim$90% relative to a fixed number of experiments with the same level of statistical detail.

Ambitiously, we envision SHARP as a way to catalyze and unite researchers and practitioners in providing reliable and reproducible performance evaluations. We provide SHARP as an open-source software that is also containerized with its prerequisites, and supports running containerized benchmarks. We plan on supporting and extending the framework and invite the community to help us in our goal.

References

[1] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, pp. 64–78, 2021.

[2] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *Transactions on Software Engineering*, vol. 47, pp. 1528–1543, July 2019.

[3] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Communications of the ACM*, vol. 59, no. 3, pp. 62–69, 2016.

[4] J. Vitek and T. Kalibera, "Repeatability, reproducibility, and rigor in systems research," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT'11, (New York, NY, USA), p. 33–38, Association for Computing Machinery, 2011.

[5] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *Transactions on Parallel and Distributed Systems*, vol. 27, pp. 3617–3630, Mar. 2016.

[6] J. Scheuner, *Performance Evaluation of Serverless Applications and Infrastructures*. PhD thesis, Chalmers University of Technology and Gothenburg University, Sweden, 2022.

[7] M. Li, J. Weng, A. Yang, W. Lu, Y. Zhang, L. Hou, J.-N. Liu, Y. Xiang, and R. H. Deng, "CrowdBC: A blockchain-based decentralized framework for crowdsourcing," *IEEE transactions on parallel and distributed systems*, vol. 30, no. 6, pp. 1251–1266, 2018.

[8] O. Novo, "Blockchain meets IoT: An architecture for scalable access management in IoT," *IEEE Internet of Things*, vol. 5, no. 2, pp. 1184–1195, 2018.

[9] A. A. Heidari, S. Mirjalili, H. Faris, I. Aljarah, M. Mafarja, and H. Chen, "Harris hawks optimization: Algorithm and applications," *Future Generation Computer Systems*, vol. 97, pp. 849–872, 2019.

[10] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, *et al.*, "A configurable cloud-scale DNN processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, IEEE, 2018.

[11] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 51–66, 2018.

[12] T. Patki, J. J. Thiagarajan, A. Ayala, and T. Z. Islam, "Performance optimality or reproducibility: That is the question," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–30, ACM/IEEE, Nov. 2019.

[13] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup, "Is big data performance reproducible in modern cloud networks?," in *symposium on networked systems design and implementation (NSDI)*, pp. 513–527, ACM, Feb. 2020.

[14] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.

[15] NAS, "Statistical challenges in assessing and fostering the reproducibility of scientific results: Summary of a workshop," 2016. https://www.ncbi.nlm.nih.gov/books/NBK350355/.

[16] T. B. Arnold and J. W. Emerson, "Nonparametric goodness-of-fit tests for discrete null distributions," *The R Journal*, vol. 3, pp. 34–39, 2011. https://doi.org/10.32614/RJ-2011-016.

[17] B. Baumer and D. Udwin, "R markdown," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 3, pp. 167–177, 2015.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *International Symposium on Workload Characterization (IISWC)*, pp. 44–54, IEEE, 10 2009.

[19] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proceedings of the international conference for high performance computing, networking, storage and analysis (SC)*, pp. 1–12, IEEE/ACM, Nov. 2015.

[20] A. B. De Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Why you should care about quantile regression," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 207–218, 2013.

[21] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The Popper convention: Making reproducible systems evaluation practical," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1561–1570, IEEE, July 2017.

[22] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, pp. 1–29, Feb. 2019.

[23] D. De Sensi, T. De Matteis, K. Taranov, S. Di Girolamo, T. Rahn, and T. Hoefler, "Noise in the clouds: Influence of network performance variability on application scalability," *SIGMETRICS Perform. Eval. Rev.*, vol. 51, p. 17–18, jun 2023.

[24] N. Buchbinder, Y. Fairstein, K. Mellou, I. Menache, and J. S. Naor, "Online virtual machine allocation with lifetime and load predictions," *SIGMETRICS Perform. Eval. Rev.*, vol. 49, p. 9–10, June 2022.

[25] L. Bulej, V. Horkỳ, P. Tuma, F. Farquet, and A. Prokopec, "Duet benchmarking: Improving measurement accuracy in the cloud," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pp. 100–107, ACM, Apr. 2020.

[26] J. Kuhlenkamp, S. Werner, M. C. Borges, D. Ernst, and D. Wenzel, "Benchmarking elasticity of FaaS platforms as a foundation for objective-driven design of serverless applications," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 1576–1585, ACM, Mar. 2020.

[27] J. Decker, P. Kasprzak, and J. M. Kunkel, "Performance evaluation of open-source serverless platforms for Kubernetes," *Algorithms*, vol. 15, no. 7, p. 234, 2022.

[28] R. Hancock, S. Udayashankar, A. J. Mashtizadeh, and S. Al-Kiswany, "Orcbench: A representative serverless benchmark," in *IEEE 15th International Conference on Cloud Computing (CLOUD'22)*, pp. 103–108, IEEE, 2022.

[29] J. Kim and K. Lee, "Practical cloud workloads for serverless FaaS," in *Proceedings of the 10th ACM Symposium on Cloud Computing (SOCC'19)*, pp. 477–477, 2019.

[30] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 2020.

[31] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SOCC'20)*, pp. 30–44, 2020.

[32] C. Abad, I. T. Foster, N. Herbst, and A. Iosup, "Serverless computing (dagstuhl seminar 21201)," in *Dagstuhl Reports*, vol. 11, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. https://drops.dagstuhl.de/opus/volltexte/2021/14798/pdf/dagrep_v011_i004_p034_21201.pdf.

[33] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A performance tester's dream or nightmare?," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pp. 138–149, ACM, Apr. 2020.

[34] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "Befaas: An application-centric benchmarking framework for FaaS platforms," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 1–8, IEEE, Oct. 2021.

[35] D. Barcelona-Pons and P. García-López, "Benchmarking parallelism in FaaS platforms," *Future Generation Computer Systems*, vol. 124, pp. 268–284, 2021.

[36] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 3–18, ACM, Apr. 2019.

[37] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units (GPGPU*, pp. 63–74, ACM, Mar. 2010.

APPENDIX

A. Abstract

The artifact contains SHARP's implementation along with the detailed instructions of running a variety of functions over a number of backends including Docker, Knative and Fission. The framework also contains Dockerfiles to build container images from sratch. The artifact can be accessed via the public repository on GitHub (https://github.com/HewlettPackard/SHARP).

B. Artifact check-list (meta-information)

- **Model:** Statistical models for stopping criteria
- **Run-time:** Docker, Knative serverless environment
- **Execution:** Rodinia function invocations with SHARP
- **Metrics:** Execution time, NAMD, KS Statistic
- **Output:** CSV logs, visualizations, statistical reports
- **Publicly available:** Yes
- **Code licenses:** MIT License
- **Archived DOI:** 10.5281/zenodo.13147448

C. Description

1) How to access: The SHARP's repository can directly be cloned from GitHub (https://github.com/HewlettPackard/SHARP).

2) Hardware dependencies: SHARP can be used to run different functions over a variety of backends and hardware. You can run Rodina or other functions in SHARP with different hardware configurations but it may not produce similar results. To reproduce the results close enough to the paper following hardware is recommended:

- **Machine 1 and 2:** AMD EPYC 7443 with Nvidia A100X 80GB GPU.
- **Machine 3:** Intel Xeon 8468V with Nvidia H100 80GB GPU.

3) Software dependencies: All the software and hardware dependencies along with installation instructions are available with the code repository and can be found here. For

4) Models: The code contains statistical models for implementing various stopping rules (Fixed, CI, KS). To reproduce the results close enough to the paper following software specification is recommended:

- **OS:** Linux 5.15.0-116-generic
- **Nvidia Driver Version:** 550.90.07
- **CUDA Version:** 12.4
- **Docker Version:** 24.0.7

D. Installation

You can clone the SHARP repository from GitHub: 'git clone https://github.com/HewlettPackard/SHARP' and follow the README instructions to set up the environment variables and configure your experiment.

E. Evaluation and expected results

The evaluation focuses on comparing the reproducibility and efficiency of different stopping criteria. The KS-based stopping rule is expected to significantly reduce the number of required runs while maintaining accurate performance distributions, as shown in our experimental results.