Variability-Guided Performance Optimization

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

Viyom Mittal
viyom.mittal@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Mohammed Baydoun
mohbay@gmail.com
American University of Beirut
Beirut, Lebanon

Aditya Dhakal
aditya.dhakal@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Izzat El Hajj
izzat.elhajj@aub.edu.lb
American University of Beirut
Beirut, Lebanon

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

Abstract

The past few decades have seen software and hardware growing more heterogeneous and layered in abstractions. This trend produced many benefits for hiding complexity and increasing efficiency and modularity. But it also makes reasoning about performance and identifying its underlying factors more challenging because of the presence of performance variability. Moreover, performance variability can prevent synchronous applications from scaling and server applications from meeting service-level agreements.

In this paper, we present a variability-guided optimization (VGO) workflow that leverages information in performance distributions to optimize performance, and more importantly, to reduce performance variability. It works by uncovering software, hardware, and compiler factors associated with specific aspects of variability and then suggesting measures for reducing it. Our experimental evaluation on a set of CPU and GPU benchmarks and applications shows that our tool successfully reduces the standard deviation and coefficient of variation of application run times by 0.374× and 0.444×, while also reducing mean run time by 0.843×. This technique enables tuning applications and their environments, improving their performance and predictability.

CCS Concepts

• **General and reference** → **Performance**; *Metrics*; Measurement;
• **Hardware** → *Hardware-software codesign.*

Keywords

Performance Evaluation, Performance Optimization, Performance Variability.

ACM Reference Format:
Eitan Frachtenberg, Viyom Mittal, Mohammed Baydoun, Aditya Dhakal, Izzat El Hajj, and Dejan Milojicic. 2026. Variability-Guided Performance Optimization. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3777884.3796994

ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3796994

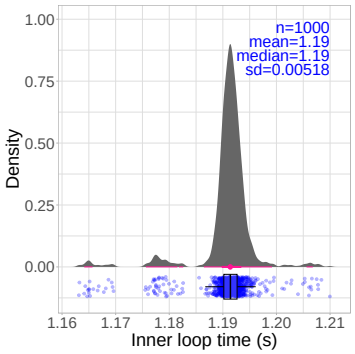


Figure 1: Run time distribution of incrementing an array of 10M integers using numpy, running 1,000 sequential repetitions on a dual Xeon 6448H server with 1TB RAM

1 Introduction

As hardware and software systems become more complex, we increasingly observe that the same code running on the same system with the same inputs exhibits different performance across runs, a.k.a performance variability [5, 12, 27, 31, 44]. System designers often optimize for average-case or best-case performance, which is sometimes traded off for higher performance variability [52].

Even the most unassuming of applications, running on a dedicated and powerful server, can experience performance variability. As a trivial example, take a single-threaded microbenchmark that increments every value of an input array of random integers once using the optimized 'numpy' library [49]. Since the array is accessed sequentially and only once, it is easy to prefetch and it doesn't suffer from memory contention, false sharing, cache misses, or branch mispredictions. As shown in Figure 1, the performance distribution of this program is mostly centered around one narrow mode, but not all runs take the same time. Although this microbenchmark runs for only a short duration, with no intentional interference and with minimal contention and resource demands, there are still numerous instances where it runs faster or slower than usual. Even under these idealized conditions, the application fails to consistently maximize attainable performance, as evidenced by the left tail of the distribution.

The upshot is that performance variability cannot be completely eliminated on modern computer systems with all their layers of abstraction and intricate subsystems. As demonstrated later in this

paper, real applications with more demanding resource requirements are even more prone to variability in their computing environment than this microbenchmark. Consequently, they exhibit higher performance variability.

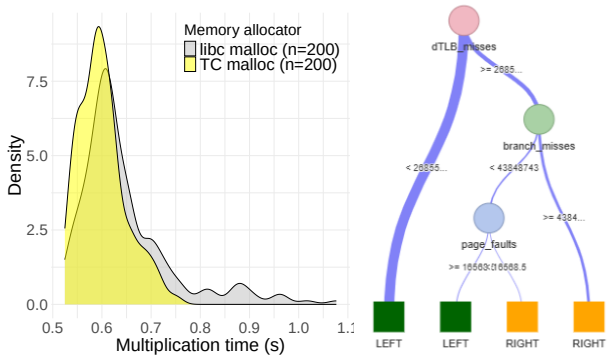
The implications of this variability can be enormous for businesses and users. For example, a business that commits to meeting certain quality-of-service (QoS) criteria or service-level agreements (SLAs) may have to overprovision compute resources at great cost to accommodate the few slow runs that would otherwise exceed these criteria [36]. However, overprovisioning does not guarantee meeting SLAs, and violations can result in high penalties [33]. As another example, tightly-coupled synchronous HPC applications may waste valuable supercomputer cycles and energy while all the synchronizing processes wait for the one process that falls on the long tail of the performance distribution [16].

Nevertheless, performance variability also hides new opportunities for performance optimization, which this work aims to expose. Traditional approaches to performance optimization focus on an application's critical path, call graphs, and hotspots. Profilers identify bottlenecked resources on the critical path of execution whose alleviation is expected to improve mean performance. However, these bottlenecked resources may not be the same factors that induce performance variability. In this paper, we propose a new and complementary approach—variability-guided optimization (VGO)—that analyzes how performance varies across repeated runs to find ways to both improve the performance and, more importantly, reduce its variability. Thus, VGO uncovers new opportunities for performance optimization that traditional approaches may miss.

Motivating Example. Let's say we wish to optimize the performance of a matrix-multiplication kernel on the CPU. To this end, we implement a simple CPU microbenchmark that squares a 25M-element matrix using 'numpy' [49]. We run it 200 times on a dedicated dual Intel E5-2680 server (48 logical cores) and find the mean run time of the multiplication kernel to be around 0.65s.

The traditional approach to performance tuning could employ a profiling tool like GNU gprof [23], which lists code hotspots where the program spends most of its time. This approach has limited utility in cases where the source code is not available or has been extensively optimized, as is the case for numpy [49]. Alternatively, specialized tools such as Intel's VTune [55] can find system-wide and microarchitectural opportunities for tuning. But what if instead of analyzing the performance of a few runs, we look at the distribution of the run time and search for interesting phenomena such as long tails or modes? What if we additionally look at distributions of low-level system metrics and try to associate factors that vary together with the application's performance? These factors might reveal different avenues for tuning the application, middleware, or hardware to reshape the performance distribution itself.

Running our matmul benchmark exhibits performance that is more variable than the increment example, with a much longer right tail (Figure 2a). To understand what happens in the system for those slow runs in the tail, we can rerun this experiment while collecting some low-level system counters, such as CPU hardware and software counters using Linux's `perf` tool [15]. Then, we can pick a point on the performance distribution, say around 0.75s, and separate all the runs left of that point as "normal" and all the runs



(a) Density plot of run time distribution using libc malloc (gray) and TC malloc (yellow). **(b) Decision tree to classify "normal" and "tail" performance.**

Figure 2: Matrix multiplication microbenchmark

right of it as "slow" or "tail". We can now fit a statistical model, such as a simple decision tree, to associate low-level metrics with the performance classes. In other words, we try to find which system factors are associated with "left" or "right" performance.

As Figure 2b shows, perhaps the most prominent factor here is dTLB cache misses: a higher miss count is associated with a higher likelihood of ending in the tail. We can try to rearrange how memory is allocated into pages to reduce page-cache misses by using a memory allocator that optimizes for tail allocation, such as Google's TC malloc [24]. Indeed, the new allocator does reduce dTLB misses, and consequently, shrinks the tail of the distribution significantly (Figure 2a). Interestingly, the mode ("fast" case) of the distribution is barely changed, but the 95^{th}-percentile latency is reduced by $\approx$ 21% and the coefficient of variation (CV) by $\approx$ 47%.

That, in a nutshell, is our proposed VGO methodology for variability reduction: we measure system-level counters alongside the application, fit a classifier to associate these counters with areas of interest in the distribution, and then try specific mitigations that address the corresponding system-level factors that influence variability. When successful, the result of this directed search process is a reduction in the variability of an application's performance, often combined with an improvement in its mean performance.

In contrast, running VTune on this benchmark using multiple analyses, including memory access and microarchitecture, surfaces no issues or recommendations regarding page faults or the TLB. This result is not surprising, considering that these metrics do not represent performance-limiting factors in most runs—only in the tail cases—which are unlikely to be randomly captured by VTune. The performance mitigations suggested by VTune have more to do with the critical path of the application (i.e., "hotspots") and not with tail performance. The main contribution of VGO is in focusing on these new opportunities for performance variability reduction that are not captured by traditional hotspot analyzers.

Overview of paper. The VGO methodology is explained in detail in Section 3, after putting our work in the context of the wide body of literature on variability in Section 2. We then experimentally validate VGO on a range of CPU and GPU applications and platforms in Section 4, where we show that it successfully reduces the standard

deviation and coefficient of variation of application performance by 0.374× and 0.444×, respectively, while also reducing mean run time by 0.843×. We follow with a discussion and elaboration of the results in Section 5. Our implementation is integrated in the open-source SHARP package[1].

2 Related Work

There is a large body of literature studying the performance variability of applications in different contexts, including hardware differences, system software differences, or dataset differences [4, 6, 7, 21, 30, 57, 63, 66]. Some works aim to model and predict the impact of these factors on performance for various purposes such as scheduling in HPC or cloud systems [2, 18, 19, 43, 47, 48, 56, 68, 69, 72]. Other studies focus on variability caused by interference from other applications [4, 6, 7, 21, 30, 63], which we plan to address in extensions to this work. But even when running with the same hardware and software configurations, the same inputs, and in the absence of interference, variability can be significant [44, 52], which surfaces the actionable insights for variability reduction that VGO exposes. This controlled performance can still be hard to capture faithfully or reproduce [10, 65] and some studies propose principles for reproducible performance evaluation [32, 50, 57], which we incorporated into our VGO implementation.

Several prior works studied specific sources of performance variability and specific mitigations for reducing the variability. Examples include addressing intra-thread variability with thread pinning, frequency control, and disabling SMT [1, 11, 12]; cache-aware physical page allocators [31]; careful task placement in large clusters [52]; controlling benchmarks for these variable factors [5]; and even synthetically inducing variability to understand its performance effects [3]. VGO attempts to reduce the large search space of such sources, which would be prohibitively expensive to search exhaustively, by directing the search to system factors that are strongly associated with performance variability.

A few other works investigated automatically detecting performance bugs and anomalies (such as PerfScope [17] and LearnConf [66]), sometimes by uncovering causal chains of events that lead to these bugs (such as the Mystery Machine [9], RESIN [41], and Pivot Tracing [42]). They differ from VGO in either methods (e.g., causal analysis, log analysis, instrumentation), objectives (e.g., finding bugs, reducing makespan in request scheduling), or domains (e.g., finding memory leaks, distributed microservices). But perhaps the key distinction is that the core of VGO is *the exploitation of system-level variability reduction opportunities revealed by repeatedly running an application in isolation.*

That said, two recent works come to mind as closer to VGO in both objectives and methods. Tuncer et al. [62] trained machine-learning models that classify runs of applications as healthy or anomalous based on a profile of performance counters. However, training these models requires a large amount of synthetically generated labeled data, unlike VGO that requires no pre-training and focuses on normal (not anomalous) variability sources. Another project, Vapro [71, 73], builds on vSensor [61] by profiling code regions used to detect variability and analyzing which performance counters influence variability the most. Our work has similar objectives to Vapro but has multiple key distinctions. First, Vapro relies on statistical analysis of the performance samples while ignoring the specifics of their distribution. Second, Vapro relies on an online approach, which limits its analysis to selected code regions and necessitates collecting only a few metrics at a time to limit the runtime overhead. Third, VGO not only reports factors associated with variability, but also suggests and sometimes automates mitigations, and is not limited to CPU-only optimization, as Vapro is.

[1]SHARP can be downloaded from https://github.com/HewlettPackard/SHARP. The version used in this paper and depicted in the appendix is V3.0.0

3 Methodology

VGO consists of our novel methodology for diagnosing and mitigating performance variability and our toolset to automate this methodology. In this section, we describe both the methodology in generalized terms as well as our reference implementation of the toolset. We have implemented the toolset as a module in the open-source SHARP framework for reproducible performance evaluation [46]. A detailed description of our workflow with screenshots can be found in Appendix 1.

Figure 3 shows the steps of the generalized methodology, which are elaborated in the rest of this section. At a high level, we start by measuring a baseline performance distribution of the application of interest and remeasure the distribution with profiling to collect performance counters (Section 3.1). Second, the distribution is divided into performance classes of interest, such as slow and fast modes or performance tails (Section 3.2). Third, we fit a model to classify each run into a performance class based on the performance counters of that run (Section 3.3). Fourth, we extract influential performance counters from the model and use them to select mitigations to apply (Section 3.4). Fifth, the mitigations are applied and the new performance distribution is measured (Section 3.5). Finally, the distribution with the mitigation applied is compared to the baseline performance distribution to assess success (Section 3.6). This process is repeated until a sufficient reduction in variability is reached or mitigations are exhausted.

3.1 Measuring Performance Distributions

Our first step is to measure a baseline performance distribution to assess the need for reducing variability and to compare all future improvements against it. This step simply consists of running the application repeatedly and collecting the performance metrics we care about, such as run time or throughput. The repeated executions should ideally run with minimal or no interference from other applications and in identical conditions across executions to capture the distribution of the application's performance alone (as in Das et al. [13]). For this purpose, we adopted the Sharp [46] open-source framework because of its focus on this kind of variability, and extended it with our techniques.

After obtaining the baseline distribution, we rerun the original application with the addition of one or more sets of low-level metric layers. Our toolset uses the recorded metadata from the baseline experiment to reproduce its configuration for the runs with counters. The system counters collect metrics that may be associated with performance variability, such as CPU counters, sensors, system-call instrumentation layers, GPU profilers, etc. We aim to exclude metrics that are either not actionable—that is, not suggestive of

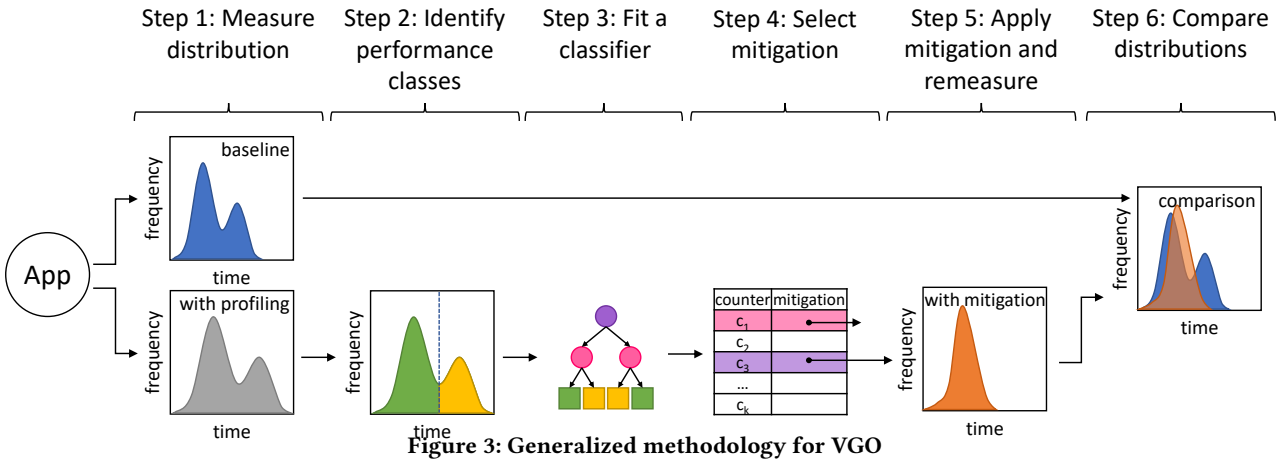


Figure 3: Generalized methodology for VGO

system or software mitigations—or are more likely consequences of the performance variability. An example of such a metric is perf's `cycles` counter: runs that are slower are often strongly correlated with a higher number of cycles because it simply means that the CPU was running for a longer time. Moreover, there are no system or software levers that we can suggest for reducing the variability in the number of cycles directly. In contrast, traditional performance-tuning tools such as VTune might be able to suggest how to reduce the average number of cycles, but that is not VGO's purpose.

3.2 Identifying the Performance Class of Interest

The next step is to inspect the distribution of performance metrics of interest: we start with application performance, such as wall-clock time, bandwidth, or any metric reported by the benchmark. As we identify influential system factors, we may also switch to inspect the distribution of those metrics, such as context switches or time spent in a specific GPU kernel. Then, we identify performance classes of interest. Typically, these classes will be the slow area of the distribution. Often, the distribution may be modal, and we may want to learn what separates one mode from another (e.g., the "fast mode" and the "slow mode"). Alternatively, the distribution may be right-tailed (with few slow runs), and we may want to know what distinguishes those runs. Sometimes, an application may even exhibit a left-tailed distribution (with few fast runs), and we may want to learn what characterizes those. Even for distributions that do not have modes or tails, we may be interested in what separates points to the left and right of the mean. In any of these cases, we decide what area of the distribution to focus on and label individual runs based on whether they fall inside or outside that area. VGO can even automatically search for a cutoff point that minimizes the entropy of the resulting labeling and model fit.

3.3 Fitting a Classifier for the Performance Class

In the final preparation phase, we fit a statistical model to classify runs using the low-level metrics based on these labels. Although the ideal model should be explainable, it need not be complex. Our goal is not to predict the performance class of new runs, but rather to determine which low-level factors are strongly associated with the class in existing runs. For example, variability in the number of page faults could be associated with variability in performance, such that a high number of faults could associate with a measurement that falls into the "slow" performance class. It is important to keep in mind that while these factors are statistically linked with performance variability, they might not cause it. However, for the purpose of reducing variability, causality is not a requirement: reducing the associated factor's variability might still work because it mitigates an underlying variable that connects it to the performance.

3.4 Selecting a Mitigation

We can now start iterating efficiently over various mitigations and measure their effect on the performance distribution. From the model we fit, we extract the features (system factors or counters) that are most strongly linked to the outcome metric. Depending on the type of model, this step can be as simple as picking the features with the largest coefficients (in a linear model) to as complex as permutation feature importance. As always, finding an association between the variability of a factor and the variability of overall performance does not necessarily imply causation. But again, even without a direct causal link, for example, due to an underlying hidden variable that affects both, reducing variability for the factor could reduce it for the application as well.

Each feature can point to a different system behavior that is affecting performance. It is worth repeating that, unlike traditional profiling, this methodology exposes features associated with a particular region of the performance distribution, not overall performance bottlenecks. For example, a high cache-miss count that is associated with long-tail performance could suggest explanations such as *specific runs* with either: high interference from kernel daemons; a higher number of core migrations; misaligned cache lines, etc. Each such explanation then suggests different mitigations (changes to the software, middleware, or hardware) to reduce variability for this factor, and consequently, for the application. For the previous three examples, these mitigations might include pinning system processes to their cores, disabling NUMA balancing for the application, or replacing libc's malloc with a different memory allocator. More examples of common factors and mitigations are shown in Table 1.

Table 1: Examples of commonly observed factors affecting performance distributions and suggestions for actions to modify them.

Factor	Potential mitigations
Context switches	Change scheduler [14]; pin threads
Thread affinity	Bind processes [28]
Branch mispredictions	Instruction hints [26]; optimize code [25]
Cache misses	Prefetching [20]; reduce interference [37] disable Simultaneous Multithreading
CPU migrations	Bind threads [58]; change scheduler; disable NUMA balancing
Page faults	Change allocator [22]; optimize code
dTLB misses	Change allocator; use huge pages [45]
Memory refresh rate	Decrease memory refresh rate in BIOS
Emulation faults	Recompile code for target architecture
L1 icache misses	Recompile with "-Os"
CPU frequency	CPU cooldown; reduce interference; Disable Processor C-State Control; BIOS Workload Profile; disable DVFS
Remote memory & Neighbor MCM Cache accesses	Enforce NUMA binding [40]
GPU memory copy	Used pinned memory
GPU temperature	Increase fan speed; frequency capping
GPU frequency	Increase fan speed; frequency capping
GPU kernel launch overhead	Use CUDA streams; use CUDA graphs

Understanding the effects of these system factors on performance and their potential mitigations requires experience and expertise, as is the case for performance tuning in general. However, VGO augments the user's human expertise in two helpful ways.

First, VGO comes prepackaged with dozens of these explanations and potential mitigations encoded in configurable YAML files. Second, it offers a simple user interface to query a local large-language model (LLM) for its explanations and mitigations. Future versions could also associate each factor with specific hotspots in the code (when accessible) for tailored advice to mitigate each factor at the source-code level.

3.5 Apply the Mitigation and Remeasure

Having selected a mitigation to try, the user must then apply the mitigation to the system and remeasure the performance distribution (without reprofiling system factors). Some of these mitigations may be as simple as changing a system-level parameter. In this case, mitigation mini-scripts are embedded in VGO's YAML configuration files and are automatically applied by VGO before remeasuring the new performance distribution. More complex mitigations may require rebooting, relinking the code, or even modifying it. In this case, the user applies the mitigations and notifies VGO when it can proceed to remeasure the mitigated distribution.

3.6 Comparing Baseline and Mitigated Distributions

Finally, we compare the original baseline performance distribution with the one obtained after the mitigation. VGO presents both a graphical summary of the two distributions overlaid on each other and a list of statistical summaries, such as the mean, median, modes, and various dispersion measures of both distributions. This comparison lets us decide whether the mitigation was successful in reshaping the distribution to our needs (e.g., reducing the tail, eliminating modes, shrinking the variance, or improving the mean). If so, we may decide to stop here. If not, we can try a different mitigation (i.e., return to Step 4) until we have exhausted VGO's suggestions. If the mitigation was successful but further improvements are desired, we can restart the process (i.e., return to Step 1) until the desired level of variability is reached.

3.7 Limitations

These steps encompass the complete methodology to optimize and reshape performance distribution based on variability analysis. Note that, like in all performance tuning efforts, one may quickly reach a point of no improvement or diminishing returns for a well-tuned system and software.

There are additional factors that could limit the applicability of VGO. Some applications or environments may not be conducive to low-level metric collection, for example virtual machines. Perhaps the most limiting factor, however, is the length of the approach. Although the search for mitigations is directed for highest impact first, repeating the execution of an application hundreds of times for several mitigations can consume significant time and computing resources. Most of the benchmarks studied in the evaluation section take under a minute to complete a single run, so these repetitions did not impose a prohibitive constraint. But for applications that run for a long time, other strategies need to be employed. For example, VGO could be run on a subset of the application's execution or a representative proxy application. Alternatively, if the application has many iterations with equal load, VGO can adopt the approach of similar studies [61, 70, 73] that treat each iteration of the application as a sample in the analysis. We demonstrate this approach on one long-running application in Section 5. When none of these approaches is feasible, traditional tuning tools may be the only choice remaining for performance optimization.

The type of application is also more consequential if we include interactive and I/O-bound applications, which introduce new sources of performance variability. Although outside the scope of this paper, it is straightforward to add the appropriate low-level metrics for these use cases to collect, extending the VGO approach so it can be applied to these applications as well.

4 Experimental Evaluation

4.1 Experimental setup

Table 2 summarizes the main experimental platforms and the benchmarks that we ran on each platform. We selected benchmarks that were found to exhibit interesting performance variability patterns and were successfully optimized by our methodology. With regards to the number of repetitions for each benchmark, VGO is built on SHARP [46], which includes features to try to determine, using statistical rules, when the performance distribution is representative enough and stop the experiment early. We opted not to use these features and instead err on the conservative side of running too many experiments, to maximize our confidence that we have captured representative distributions. However, for practitioners

looking to use VGO to optimize real applications, the use of such time-saving features is encouraged.

Table 2: Characteristics of evaluation servers and benchmarks executed on them. All servers are dual-socket servers.

Server	CPU	Cores (Threads)	Memory	GPU	Benchmarks
Server-1	2× Intel Xeon 6448H	64 (128)	1 TB	-	7z [29], lbm [60], sad [60]
Server-2	2× AMD EPYC 7443	48 (96)	270 GB	-	Xapian [67], LavaMD [8]
Server-3	2× AMD EPYC 7601	64 (128)	2TB	A100	srad-cpu [8]
Server-4	2× AMD EPYC 7443	48 (96)	270 GB	A100	gpt-oss-20b [51], srad-gpu [8], Yi-6B [35], Retinanet [39], Stable diffusion [59]

4.2 Results overview

Table 3 summarizes our experimental results. For each benchmark, the table shows the influential performance factor identified by VGO, the corresponding mitigation applied, and the reduction in mean run time, standard deviation (SD), and coefficient of variation (CV) as a result of applying the mitigation. While all benchmarks experience a reduction in mean latency (0.843× geomean), we note that this reduction is not our main objective but only a side effect of reducing variability. We are more interested in the reduction achieved in the SD by all benchmarks (0.374× geomean), which reflects our methodology's success in reducing variability.

Even some of the improvements in SD can be biased by the improvement in mean run time, since shorter-running applications may vary for a smaller absolute time duration. To account for this effect, we also report the CV, which normalizes the SD to the mean, giving the variation as a ratio of the mean. Notably, even when considering the CV, all benchmarks exhibited a reduction in this metric (0.444× geomean). This result demonstrates the effectiveness of our methodology in not only improving performance but, more importantly, reducing the variability of the performance relative to the mean. For the rest of this section, we explore each benchmark in detail to understand the impact of variability reduction on that benchmark and extract relevant insights.

4.3 7z

We run 7z [29] single-threaded on Server-1. Using our methodology, we find that the performance distribution has a long tail, with context switches prominently separating the tail from the mode. To mitigate the impact of context switches, we apply the suggested mitigation of pinning threads (one in this case) to their cores. Thread pinning causes threads to return to the same core they were previously on after being context-switched, thereby finding their data still in the cache, which lowers the variability of context switching.

Figure 4a shows the performance distribution of 7z before and after thread pinning. It is clear that thread pinning results in a narrower performance distribution and a significantly shorter tail. Note that the mode and mean performance are not significantly affected by the optimization because that is not the objective of our methodology (the mean is only reduced by 0.995×). On the other hand, the SD and CV are significantly reduced by 0.159× and 0.160×, respectively. In other words, we have optimized primarily for predictable performance (dispersion) rather than mean performance.

4.4 lbm

We run lbm [60] with the *long* dataset on Server-1. Similar to 7z, our methodology identifies that the performance distribution has a long tail, and that a prominent factor separating the tail from the modes is context switches. To mitigate the impact of context switches, we apply the suggested mitigation of pinning threads to their cores.

Figure 4b shows the performance distribution of lbm before and after thread pinning is applied. The comparison shows that the optimization reduces the running time of the benchmark, but more importantly, reduces its variability. While the mean run time is reduced by 0.840×, the SD is reduced by 0.339× and the CV by 0.403×. Again, this result shows the effectiveness of our methodology in finding optimizations that reduce variability.

4.5 sad

We run the sad [60] benchmark with the *large* dataset on Server-1. We do not identify any distinct performance classes such as modes or tails. Nevertheless, by separating runs that are left and right of the mean, we find that the variability is most associated with TLB misses and suggests using transparent huge pages as a mitigation.

Figure 4c shows the performance distribution of sad before and after transparent huge pages are applied. The optimization reduces the mean running time of the benchmark by 0.941×, but more importantly, it reduces the SD by 0.833× and the CV by 0.885×.

4.6 Xapian

We run Xapian [67] with auto-generated queries using Zipfian distribution (s=0.99) for realistic query patterns on Server-2. It is an interesting case for variability reduction optimizations because it is sensitive to tail performance, and is included in the TailBench++ suite [38]. Running Xapian with our methodology, we identify that the performance distribution indeed has a long tail and two modes, and that a prominent factor separating the tail from the modes is CPU migrations. To reduce its impact, we apply the suggested mitigation of disabling NUMA balancing.

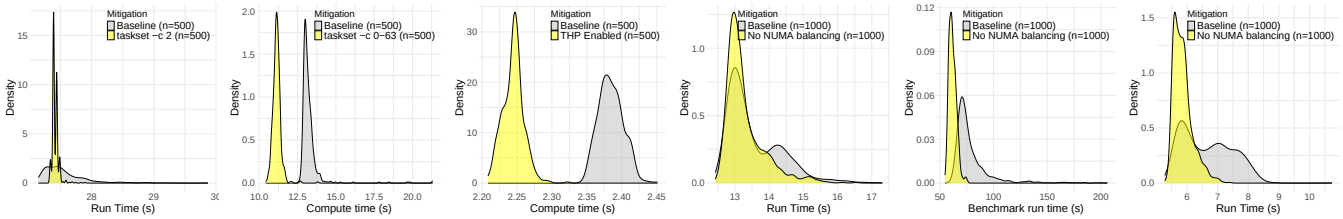
Figure 4d shows the effect of this mitigation. The optimization pushes the slow mode towards the fast mode and shrinks the tail, while keeping the fast mode unchanged. Consequently, the SD is reduced by 0.728× and the CV by 0.742×. Despite the fast mode staying put, the mean running time is reduced by 0.981× due to the reduction of the slow mode and the tail. Once again, this result demonstrates the effectiveness of our methodology at identifying optimizations that reduce variability, despite not specifically targeting mean performance. It is noteworthy that workloads like search engines may have established memory access patterns, so disabling dynamic NUMA balancing enables the efficient use of these access patterns and avoids unnecessary CPU migrations.

4.7 srad (CPU version)

We run srad [8] with the parameters '60000 0.5 502 458 128' on Server-3. Similar to Xapian, our methodology identifies that the performance distribution has a long tail, and that a prominent factor separating the tail from the mode is CPU migrations. We

Table 3: Influential factors identified, mitigations applied, and run time variability reduction achieved for each benchmark.

HW	Benchmark (release year)	System factor	Mitigation	Before mitigation			After mitigation			Reduction ratio		
				Mean	SD	CV	Mean	SD	CV	Mean	SD	CV
CPU	7z (2016)	Context switches	Pin threads	27.553	0.352	0.013	27.416	0.056	0.002	0.995×	0.159×	0.160×
	lbm (2012)	Context switches	Pin threads	13.258	0.803	0.061	11.141	0.272	0.024	0.840×	0.339×	0.403×
	sad (2012)	TLB misses	Transparent huge pages	2.385	0.018	0.008	2.245	0.015	0.007	0.941×	0.833×	0.885×
	Xapian (2011)	CPU migrations	Disable NUMA balancing	13.542	0.786	0.058	13.286	0.572	0.043	0.981×	0.728×	0.742×
	srad (2009)	CPU migrations	Disable NUMA balancing	78.630	17.541	0.223	61.810	3.500	0.057	0.786×	0.200×	0.254×
	lavaMD (2009)	CPU migrations	Disable NUMA balancing	6.657	0.817	0.123	5.869	0.318	0.054	0.882×	0.389×	0.441×
		Cache misses	CPU with larger cache	6.657	0.817	0.123	3.465	0.037	0.011	0.521×	0.045×	0.087×
		Cache misses	Disable SMT	3.465	0.037	0.011	3.454	0.024	0.007	0.997×	0.649×	0.651×
GPU	srad (2009)	Kernel launch overhead	Optimize streams usage	42.645	0.625	0.015	20.065	0.086	0.004	0.471×	0.138×	0.292×
	Retinanet (2018)	CUDA initialization	Avoid re-initialization	16.381	1.395	0.085	11.755	0.480	0.041	0.718×	0.344×	0.479×
	Stable diffusion (2023)	Temperature	Increase fan speed	6.505	0.032	0.005	6.438	0.030	0.005	0.990×	0.938×	0.947×
		Temperature	Frequency capping	6.438	0.030	0.005	6.422	0.026	0.004	0.998×	0.867×	0.869×
	gpt-oss-20b (2025)	Frequency	Frequency capping	19.516	0.353	0.018	19.469	0.160	0.008	0.998×	0.453×	0.454×
	Yi-6B (2023)	Frequency	Frequency capping	20.553	0.477	0.023	20.433	0.391	0.019	0.994×	0.820×	0.824×
Geometric mean										**0.843×**	**0.374×**	**0.444×**



(a) 7z (thread pinning) (b) lbm (thread pinning) (c) sad (transparent huge pages) (d) Xapian (disabling NUMA balancing) (e) srad (disabling NUMA balancing) (f) lavaMD (disabling NUMA balancing)

Figure 4: Performance distribution for each CPU application with mitigation in parentheses

again mitigate the impact of CPU migrations by disabling NUMA balancing as suggested.

Figure 4e shows the impact of NUMA balancing on srad. The mitigation visibly shrinks the tail and even slightly improves the mode run time. As a result, the mean runtime is reduced by 0.786×, but more importantly, the SD is reduced by 0.200× and the CV by 0.254×.

4.8 LavaMD

We run LavaMD [8] with parameters '-cores 16 -boxes1d 32' on Server-2.[2] Our methodology identifies that the performance distribution is wide with two modes, and that a prominent factor separating the modes is again CPU migrations. We mitigate the impact of CPU migrations by disabling NUMA balancing as suggested.

Figure 4f shows the performance distribution of lavaMD before and after disabling NUMA balancing. It is evident that the mitigation fuses the slower mode into the faster mode and narrows the distribution, without affecting the performance of the faster mode. Consequently, the mean runtime is reduced by 0.882×, but more importantly, the SD is reduced by 0.389× and the CV by 0.441×. Note that since LavaMD is an old application, it is not designed to utilize NUMA balancing to its advantage, so disabling NUMA balancing makes the memory access simpler.

Another prominent factor that we identified was cache misses. One suggestion to mitigate this factor is to use a CPU with a larger cache capacity. Figure 5 shows the performance distribution of LavaMD when using Server-1, a dual socket Intel with 2× the L1 data cache, 1.33× the L1 instruction cache, and 5× the L2 cache compared to the baseline AMD Server-2. Although mean run time

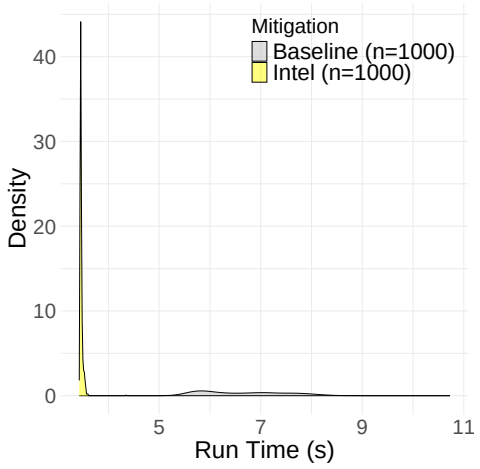


Figure 5: LavaMD performance when using CPU with more cache

cannot be meaningfully compared when switching architectures, it is worth noting the significant reductions in SD and CV, by factors of 0.045× and 0.087× respectively.

To shrink the tail even further, we applied our methodology to lavaMD again, but this time on the Intel CPU. Interestingly, the prominent factor separating the tail was identified to be cache misses. Although we could run on a CPU with an even larger cache, we instead took another suggestion to reduce cache misses, which was to disable SMT, thereby reducing cache contention between threads. Figure 6 shows the performance distribution of lavaMD on

[2]We use 16 cores because the program fails to run with a higher number.

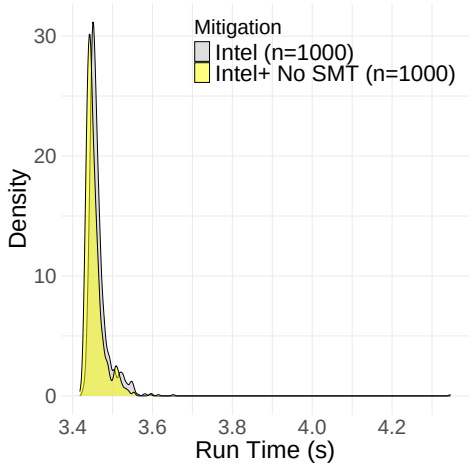


Figure 6: LavaMD performance when disabling SMT on CPU with more cache

the dual socket Intel Xeon-6448H CPU before and after disabling SMT. The mean run time is negligibly impacted and only reduced by 0.997×; however, the SD is reduced significantly by 0.649× and the CV similarly by 0.651×.

4.9 srad (GPU version)

We now turn our attention to GPU benchmarks. We run srad [8] on an NVIDIA A-100 GPU (Server-3) using the parameters '1000000 0.5 502 45'. The resulting performance distribution exhibits a tail, with one prominent factor associated with the variability: CUDA kernel launch overhead. The suggested mitigation for reducing the variability of the launch overhead is to use streams to overlap the kernel launch overhead with the execution of preceding kernels.

This optimization cannot be applied automatically and requires code inspection and modification. We noticed that the GPU kernels are executed for multiple iterations, but are synchronized every iteration with a memory copy from the GPU memory to the host memory to perform some thread-invariant computations on the CPU. We modified the code to perform that computation redundantly across threads on the GPU instead, eliminating the disruptive memory copy and allowing all kernels in all iterations to be submitted to a stream off the critical path of execution.

Figure 7a shows the performance distribution of srad on the GPU before and after removing the disruptive memory copy. It is clear that the optimization improves performance as expected, but also reduces variability. The mean runtime is reduced by 0.471×, but more importantly, the SD is reduced by 0.138× and the CV by 0.292×.

4.10 Retinanet

We run Retinanet [39], which is available in the MLPerf inference benchmark [54], on Server-4 to infer a batch of 8 images from OpenImages [34]. Our methodology identifies that the performance distribution has two modes, and that the prominent factor that separates the modes is the CUDA initialization (i.e., `cudaInit`) and context creation overhead. Although we did not have built-in suggested mitigations for this source of variability, we naturally concluded that we must avoid frequent initializations. Upon inspecting the application code, we observed that the baseline application restarts the inference process every time a new batch of images is ready. The mitigation we applied is to keep the process alive across batches, thus retaining the CUDA context when performing inference on the next batch of images. Only the initialization part was modified, and the model and data loading parts remained unchanged.

Figure 7b shows the performance distributions before (per-inference initialization) and after (single initialization) our optimization. The mean run time is reduced by 0.718×, but more importantly, the variability was significantly reduced, whereby the two modes were brought together, the SD was reduced by 0.344×, and the CV was reduced by 0.479×.

4.11 Stable diffusion

We run Stable Diffusion XL [59] on Server-4. The workload generates 1024×1024-pixel images from the prompt "Image of cat cooking steak" using 50 inference steps, guidance scale 7.5, and FP16 precision. Our methodology identifies that the distribution has two modes and a left (fast) tail, and that the factors distinguishing the fast tail and mode from the larger slow mode were the repeat iteration number and the device temperature. In other words, as the repeat count increased, the device got hotter and performance became slower due to throttling. The suggested mitigations for device temperature were increasing the fan speed and capping the frequency.

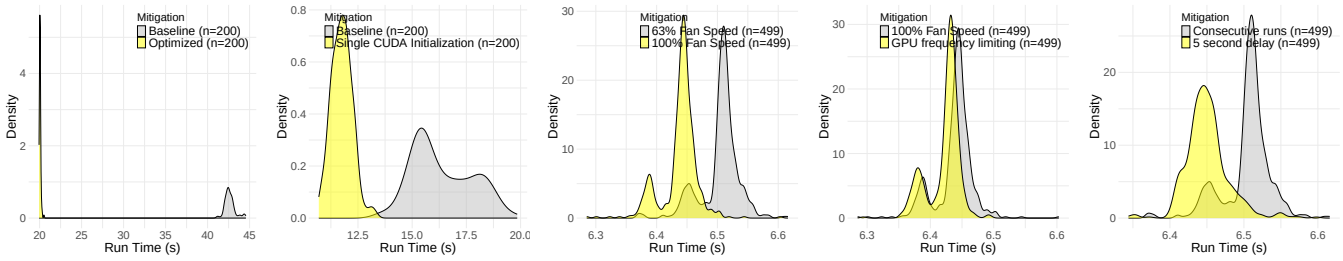
Figure 7c shows the performance distribution of stable diffusion before and after increasing the fan speed. Aside from the mean run time being reduced by 0.990×, the SD was reduced by 0.938×, and the CV was reduced by 0.947×. To further reduce variability, we applied frequency capping as well. Figure 7d shows the performance distribution of stable diffusion before and after frequency capping. The mean run time was further reduced by a negligible 0.998×; however, the SD was more notably reduced by 0.867×, and the CV by 0.869×.

Unfortunately, despite these two optimizations and the reductions in variability, the two modes persisted. To confirm that the device temperature was indeed the likely cause of modality, we introduced a 5s delay between runs to allow the device to cool down. Figure 7e shows that this mitigation indeed fuses the slow mode into the fast mode. However, it may not be desirable in realistic deployments, because it trades off throughput for latency. Nevertheless, this result corroborates that our methodology was indeed able to identify the likely cause of the two modes, despite the difficulty of eliminating the slow mode in practice.

4.12 gpt-oss-20b

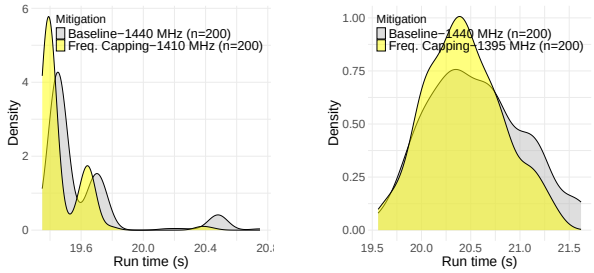
We run the LLM gpt-oss-20b, available through HuggingFace in PyTorch [51], on Server-4 with a set of five generic questions for prompts. Our methodology reveals that the performance distribution has multiple modes and a long tail, with the clock frequency being the distinguishing factor, and recommends frequency capping as a mitigation.

Figure 8a shows the performance distribution of gpt-oss-20b before and after frequency capping, where the frequency cap is



(a) srad (eliminate disruptive mem copy) **(b) Retinanet (single CUDA initialization)** **(c) stable diffusion (increasing fan speed)** **(d) stable diffusion (frequency capping)** **(e) stable diffusion (5-second delay)**

Figure 7: Performance distribution for each GPU application with mitigation in parentheses



(a) gpt-oss-20b **(b) Yi-6B with vLLM**

Figure 8: Performance distribution for LLMs on GPU with frequency capping mitigation

reduced from the maximum 1440 MHz to 1410 MHz. Although mean performance is negligibly reduced by 0.998×, variability is improved with the tail shrunk and the SD and CV reduced by 0.453× and 0.454×, respectively.

4.13 Yi-6B

We run the LLM Yi-6B, available through HuggingFace, this time using the vLLM [35] serving framework, on Server-4. Our methodology reveals a more symmetric performance distribution. Still, by selecting the runs to the left and right of the median as our performance classes, our methodology again reveals that clock frequency is a prominent factor associated with slow performance, despite the large difference in distribution shape.

Figure 8b shows the performance distribution of Yi-6B before and after frequency capping, where the frequency cap is reduced from the maximum 1440 MHz to 1395 MHz. Although mean performance is negligibly reduced by 0.994×, the distribution becomes narrower, and the SD and CV are reduced by 0.820× and 0.824×, respectively.

5 Discussion

5.1 Optimizing performance vs. reducing variability

A recurring observation from our evaluation is that reducing variability and optimizing mean performance are not identical. While some of our benchmarks indeed experience an improvement in mean performance (e.g., lbm, sad, srad, lavaMD, Retinanet), others experience a reduction in variability while the mean or mode performance remained mostly unchanged (e.g., 7z, Xapian, stable diffusion, gpt-oss-20b, Yi-6B). This distinction highlights the need for a systematic methodology for diagnosing and mitigating performance variability beyond traditional profiling. With traditional profiling, if the programmer happens to profile runs that are in the fast mode or off the tail, they may miss optimization opportunities. By observing runs across multiple modes or on and off the tail, and by identifying the factors that distinguish these runs, our methodology is capable of specifically diagnosing the source of variability and recommending mitigations to decrease it.

5.2 Factors before mitigations

In our methodology, we start by identifying the system factors associated with the performance variability, then recommend mitigations. This approach is essential because the brute-force approach of exhaustively applying all possible mitigations without guidance is expensive. For example, a common recommendation for reducing variability is to disable SMT [5, 11, 12]. However, Figure 9 shows that disabling SMT increases the variability of the lbm benchmark. In particular, it increases the SD and CV by 1.10× and 1.13×, respectively. Therefore, a guided approach such as ours for reducing variability is necessary. Indeed, VGO does not recommend this mitigation for this application. Interestingly, in this example, disabling SMT reduces the mean run time by 0.974× despite increasing variability. This observation reiterates our earlier point that optimizing performance and reducing variability are different and sometimes opposite objectives that require different methodologies.

5.3 Long-running applications

Long-running applications introduce multiple challenges for variability analysis: they tend to have multiple phases, resulting in loss of information when averaging metrics across phases; and they are impractical to run a large number of times. To demonstrate VGO's usage in such applications, we apply it to a video segmentation program based on Segment Anything Model 2 (SAM2) [53]. The program consists of multiple phases, including copying a video to main memory, decoding it in main memory using the CPU, transferring it to the GPU memory and segmenting it on the GPU, and writing the outputs back to storage. The program pipelines these operations across consecutive videos sent to the same GPU, and executes multiple such pipelines in parallel across multiple GPUs.

We ran the program on a compute node having a 64-Core AMD EPYC 7763 CPU and 8 Nvidia A100 GPUs. Each video has 5K resolution (5120×2880) and 483 frames. Since the application has multiple

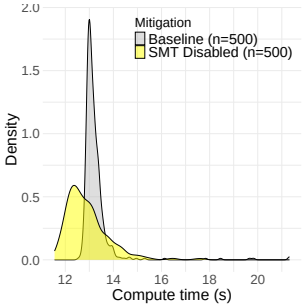


Figure 9: Impact of SMT on lbm

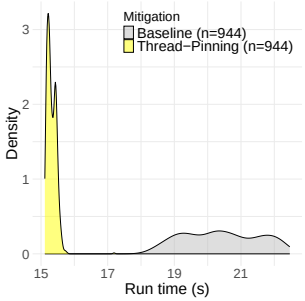


Figure 10: Impact of pinning on SAM2

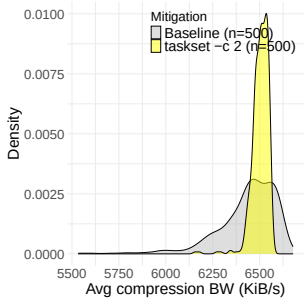


Figure 11: Impact of pinning on 7z

phases, we instrumented and collected data for each phase separately as well as the application as a whole. Moreover, since the application executes many videos in parallel across multiple GPUs as well as in pipeline on the same GPU, we treated each video as a single observation to gather many observations from a single run. After collecting the data, we applied VGO's classification stage to the end-to-end latency using the performance of the individual phases as factors.

We found that the variability in the end-to-end latency was most associated with the variability in the decode phase running on the CPU. Note that this phase is not the longest-running one, but rather the one with the most impact on variability. We then applied VGO to the data collected in the decoding phase and our methodology revealed a distribution with three modes, with context switches being the distinguishing factor of the slowest mode and thread pinning being the recommended mitigation.

Figure 10 shows the latency distribution of the decode phase before and after we apply selective thread pinning to the subset of CPU threads performing the decoding. The mean latency is subsequently reduced by approximately 0.724×, but more importantly, the CV is reduced to 0.037× and the SD to 0.027×, an impressive reduction in variability. This reduction in variability of the decode phase aggregated across many videos results in a 3.5% end-to-end increase in application throughput (in frames per second).

5.4 Diversity of performance distributions

We have observed that performance distributions can have diverse shapes with different benchmarks having multiple modes (e.g., Xapian, lavaMD, Retinanet, stable diffusion, gpt-oss-20b) or long tails (e.g., 7z, lbm, Xapian, srad, gpt-oss-20b). Even for Gaussian-looking distributions that do not have any modes or tails (e.g., sad, Yi-6B), analysis of variability around the median can yield fruitful mitigations for making the distribution narrower in general. This diversity in the shape of distributions demonstrates the need for a flexible methodology for reducing variability that is aware of the shape of the distribution and can focus on different performance classes within the distribution, such as what VGO provides.

5.5 Generalizing to different platforms and metrics

Our tool can work on a variety of platforms and with a variety of profiling tools. We have seen examples of single-threaded CPU applications (e.g., 7z); multi-threaded CPU applications with metrics collected from tools such as Linux `perf`; GPU applications with metrics collected from tools such as `nvidia-smi` and `nsys`; and a combination of CPU and multiple GPUs using a combination of these profiling tools. The VGO approach can additionally scale to analyze multi-node performance variability. We have implemented initial MPI instrumentation using `mpip` [64] to uncover network-related factors. In future work, we plan to investigate the variability of individual ranks in HPC applications at scale.

The VGO approach is also agnostic to which performance metric we are trying to optimize, or even to whether we are trying to minimize or maximize it. As an example, the 7z benchmark from Section 4.3 reports different performance statistics, and in Figure 11, we see the results of collecting and analyzing the average compression bandwidth instead of run time. Since this metric represents higher performance with higher values, it is nearly a mirror image of Figure 4a, with a left tail instead of right. But VGO works just as well identifying the same factor and mitigation for this metric, and the consequent thread pinning results in a similar (but slightly smaller) improvement in performance. In other examples, outside the scope of this paper, we also collected metrics for energy use and power efficiency, response time, I/O time, and other useful metrics.

5.6 Generalizing across time

In our experiments, older legacy applications such as Rodinia benchmarks, Xapian, and sad achieve better performance when recent system features such as Transparent Huge Pages and NUMA Balancing are disabled (see Table 3 for release years). In contrast, newer workloads such as Retinanet, Stable Diffusion, and LLMs are well-tuned for modern systems. For these applications, performance gains come mostly from hardware and system-focused optimizations, including GPU frequency and fan speed tuning and lowering hardware/system initialization costs. With VGO's approach, we are able to pinpoint the key performance factors and suggest mitigation for improving runtime and variance across both legacy and modern workloads.

6 Conclusion

The primary contribution of this work is the introduction of a methodology for performance and variability optimization based on information found in the performance distribution. Typical performance tuning methods already produce likely factors in bottlenecks from static or within-application variability, but VGO exposes new opportunities for optimization in the shape of the distribution and

in infrequent events. Additionally, a focus on the distribution allows the optimization not only of summary statistics like mean and 95^{th}-percentile performance, but also reshaping the distribution itself and reducing its variance.

A secondary contribution is the integration of VGO within an open-source tool that automates most of the methodology, including the collection of performance distributions, system-level metrics, identifying areas of interest, fitting a classifier for these areas, exposing influential factors, suggesting mitigations, and comparing the mitigated distributions to the original. The tool includes a knowledge base with dozens of well-known system-level factors and suggested mitigations that are nevertheless finding new application in relation to variability. Moreover, in some cases, the tool can even apply the system mitigations itself, without human labor.

Finally, The methodology and the software that embodies it are both flexible and extensible. It is trivial to add new layers of system metrics to collect, application metrics to optimize for, new factors and mitigations, and hardware backends and platforms to run on, such as CPUs, GPUs, storage arrays, and MPI environments.

References

[1] Bilge Acun. 2017. *Mitigating variability in HPC systems and applications for performance and power efficiency.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 469–482. https://www.usenix.org/system/files/conference/nsdi17/nsdi17-alipourfard.pdf

[3] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. 2019. HPAS: An HPC performance anomaly suite for reproducing performance variations. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[4] Rohan Basu Roy, Vijay Gadepally, and Devesh Tiwari. 2025. DarwinGame: Playing Tournaments for Tuning Applications in Noisy Cloud Environments. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 264–279.

[5] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2019. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21 (Feb. 2019), 1–29. doi:10.1007/s10009-017-0469-y

[6] Lubomír Bulej, Vojtěch Horkỳ, and Petr Tuma. 2019. Initial experiments with duet benchmarking: Performance testing interference in the cloud. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 249–255.

[7] Lubomír Bulej, Vojtěch Horkỳ, Petr Tuma, François Farquet, and Aleksandar Prokopec. 2020. Duet benchmarking: Improving measurement accuracy in the cloud. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 100–107. doi:10.1145/3358960.3379132

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54.

[9] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 217–231.

[10] Christian Collberg and Todd A Proebsting. 2016. Repeatability in computer systems research. *Commun. ACM* 59, 3 (2016), 62–69.

[11] Minyu Cui, Nikela Papadopoulou, and Miquel Pericàs. 2023. Analysis and characterization of performance variability for openmp runtime. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 1614–1622.

[12] Minyu Cui and Miquel Pericas. 2025. Characterizing and Mitigating Performance Variability in Parallel Applications on Modern HPC multicore Systems. In *Proceedings of the 22nd ACM International Conference on Computing Frontiers*. 151–158.

[13] Anwesha Das, Daniel Ratner, and Alex Aiken. 2022. Performance variability and causality in complex systems. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 19–24.

[14] Ajoy K Datta and Rajesh Patel. 2013. Cpu scheduling for power/energy management on multicore processors using cache miss and context switch data. *Transactions on Parallel and Distributed Systems* 25, 5 (May 2013), 1190–1199. doi:10.1109/TPDS.2013.148

[15] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[16] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefler. 2022. Noise in the clouds: Influence of network performance variability on application scalability. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 3 (Dec. 2022), 1–27. doi:10.1145/3570609

[17] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.

[18] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Notices* 48, 4 (April 2013), 77–88. doi:10.1145/2499368.2451125

[19] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Notices* 49, 4 (April 2014), 127–144. doi:10.1145/2644865.2541941

[20] James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*. 68–75. doi:10.1145/263580.263597

[21] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. 2020. Microservices: A performance tester's dream or nightmare?. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 138–149. doi:10.1145/3358960.3379124

[22] Yi Feng and Emery D Berger. 2005. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 workshop on Memory system performance*. ACM, 68–77. doi:10.1145/1111583.1111594

[23] Jay Fenlason and Richard Stallman. 1988. GNU gprof. *GNU Binutils. Available online: http://www. gnu. org/software/binutils (accessed on 21 April 2018)* (1988).

[24] Tais B Ferreira, Rivalino Matias, Autran Macedo, and Lucio B Araujo. 2011. An experimental study on memory allocators in multicore and multithreaded applications. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 92–98.

[25] Agner Fog. 2023. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.* Copenhagen University College of Engineering. https://www.agner.org/optimize/microarchitecture.pdf

[26] Rao Fu, Jiwei Lu, Antonia Zhai, and Wei-Chung Hsu. 2006. A study of the performance potential for dynamic instruction hints selection. In *Advances in Computer Systems Architecture: 11th Asia-Pacific Conference, (ACSAC'06)*. Springer, Shanghai, China, 67–80.

[27] Joshua Garland, Ryan James, and Elizabeth Bradley. 2013. Determinism, complexity, and predictability in computer performance. *arXiv preprint arXiv:1305.5408* (5 2013). https://arxiv.org/pdf/1305.5408.pdf

[28] gcc. [n. d.]. OMP_PROC_BINDING. https://gcc.gnu.org/onlinedocs/libgomp/OMP_005fPROC_005fBIND.html

[29] Bo Han and Bolang Li. 2016. Lossless Compression of Data Tables in Mobile Devices by Using Co-clustering. *International Journal of Computers Communications & Control* 11, 6 (2016), 776–788.

[30] Sören Henning, Adriano Vogel, Esteban Perez-Wohlfeil, Otmar Ertl, and Rick Rabiser. 2025. When Should I Run My Application Benchmark? Studying Cloud Performance Variability for the Case of Stream Processing Applications. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 400–410.

[31] Michal Hocko and Tomas Kalibera. 2010. Reducing performance non-determinism via cache-aware page allocation strategies. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM, 223–234. doi:10.1145/1712605.1712640

[32] Torsten Hoefler and Roberto Belli. 2015. 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)*. IEEE/ACM, 1–12. doi:10.1145/2807591.2807644

[33] Andreas Kohne, Damian Pasternak, Lars Nagel, and Olaf Spinczyk. 2016. Evaluation of SLA-based decision strategies for VM scheduling in cloud data centers. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms* (London, United Kingdom) *(CrossCloud '16)*. Association for Computing Machinery, New York, NY, USA, Article 6, 5 pages. doi:10.1145/2904111.2904113

[34] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. 2020. The Open Images Dataset V4: Unified Image Classification, Object Detection, and Visual Relationship Detection at Scale. *International Journal of Computer Vision* 128, 7 (March 2020), 1956–1981. doi:10.1007/s11263-020-01316-z

[35] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[36] Jason Lango. 2014. Toward software-defined SLAs. *Commun. ACM* 57, 1 (Jan. 2014), 54–60. doi:10.1145/2541883.2541894

[37] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the workshop on Experimental Computer Science*. ACM, 2–es. doi:10.1145/1281700.1281702

[38] Zhilin Li, Lucia Pons, Salvador Petit, Julio Sahuquillo, and Julio Pons. 2025. TailBench++: Flexible Multi-Client, Multi-Server Benchmarking for Latency-Critical Workloads. *arXiv preprint arXiv:2505.03600* (2025).

[39] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal Loss for Dense Object Detection. arXiv:1708.02002 [cs.CV] https://arxiv.org/abs/1708.02002

[40] Linux. [n. d.]. Numactl Linux Man Page. https://linux.die.net/man/8/numactl

[41] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. 2022. {Resin}: a holistic service for dealing with memory leaks in production cloud infrastructure. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 109–125.

[42] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–28.

[43] Giovanni Mariani, Andreea Anghel, Rik Jongerius, and Gero Dittmann. 2017. Predicting cloud performance for hpc applications: A user-oriented approach. In *17th International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 524–533. doi:10.1109/CCGRID.2017.11

[44] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 409–425. https://www.usenix.org/conference/osdi18/presentation/maricq

[45] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. Mega: Overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. ACM, 121–131. doi:10.1145/3319647.3325839

[46] Viyom Mittal, Pedro Bruel, Michalis Faloutsos, Dejan Milojicic, and Eitan Frachtenberg. 2024. SHARP: A Distribution-Based Framework for Reproducible Performance Evaluation. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 82–93.

[47] Amir Nassereldine, Safaa Diab, Mohammed Baydoun, Kenneth Leach, Maxim Alt, Dejan Milojicic, and Izzat El Hajj. 2023. Predicting the Performance-Cost Tradeoff of Applications Across Multiple Systems. In *IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 216–228. doi:10.1109/CCGrid57682.2023.00029

[48] Daniel Nichols, Aniruddha Marathe, Kathleen Shoga, Todd Gamblin, and Abhinav Bhatele. 2022. Resource Utilization Aware Job Scheduling to Mitigate Performance Variability. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 335–345. doi:10.1109/IPDPS53621.2022.00040

[49] Travis E Oliphant et al. 2006. *Guide to numpy*. Vol. 1. Trelgol Publishing USA.

[50] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim Von Kistowski, Ahmed Ali-Eldin, Cristina L Abad, José Nelson Amaral, Petr Tuma, and Alexandru Iosup. 2019. Methodological principles for reproducible performance evaluation in cloud computing. *Transactions on Software Engineering* 47, 8 (July 2019), 1528–1543. doi:10.1109/TSE.2019.2927908

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[52] Tapasya Patki, Jayaraman J Thiagarajan, Alexis Ayala, and Tanzima Z Islam. 2019. Performance optimality or reproducibility: That is the question. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM/IEEE, 1–30. doi:10.1145/3295500.3356217

[53] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, et al. 2024. Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714* (2024).

[54] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 446–459. doi:10.1109/ISCA45697.2020.00045

[55] James Reinders. 2005. *VTune performance analyzer essentials*. Vol. 9. Intel Press Santa Clara.

[56] Majid Salimi Beni, Sascha Hunold, and Biagio Cosenza. 2024. Analysis and prediction of performance variability in large-scale computing systems. *The Journal of Supercomputing* 80, 10 (2024), 14978–15005.

[57] Joel Scheuner. 2022. *Performance Evaluation of Serverless Applications and Infrastructures*. Ph. D. Dissertation.

[58] Pirah Noor Soomro, Muhammad Aditya Sasongko, and Didem Unat. 2018. BindMe: A thread binding library with advanced mapping algorithms. *Concurrency and Computation: Practice and Experience* 30, 21 (June 2018), e4692. doi:10.1002/cpe.4692

[59] Stability AI. 2025. Text-to-image generation using Stable Diffusion XL (stabilityai/stable-diffusion-xl-base-1.0) with thermal management interventions on NVIDIA A100 GPU. https://huggingface.co/stabilityai/stable-diffusion-xl-base-1.0. Accessed: 2025-08-19.

[60] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[61] Xiongchao Tang, Jidong Zhai, Xuehai Qian, Bingsheng He, Wei Xue, and Wenguang Chen. 2018. Vsensor: Leveraging fixed-workload snippets of programs for performance variance detection. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 124–136.

[62] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. 2017. Diagnosing performance variations in HPC applications using machine learning. In *High Performance Computing: 32nd International Conference, ISC High Performance*. Springer, 355–373. https://www.bu.edu/peaclab/files/2020/01/isc.pdf

[63] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is big data performance reproducible in modern cloud networks?. In *symposium on networked systems design and implementation (NSDI)*. ACM, 513–527. https://www.usenix.org/system/files/nsdi20-paper-uta.pdf

[64] Jeffrey S Vetter and Michael O McCracken. 2001. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. 123–132.

[65] Jan Vitek and Tomas Kalibera. 2011. Repeatability, Reproducibility, and Rigor in Systems Research. In *Proceedings of the Ninth ACM International Conference on Embedded Software* (Taipei, Taiwan) *(EMSOFT'11)*. Association for Computing Machinery, New York, NY, USA, 33–38. doi:10.1145/2038642.2038650

[66] Yang Wang, Miao Yu, Yujie Hui, Fang Zhou, Yuyang Huang, Rui Zhu, Xueyuan Ren, Tianxi Li, and Xiaoyi Lu. 2022. A study of database performance sensitivity to experiment settings. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1439–1452.

[67] xapian.org. 2025. Xapian project. https://github.com/xapian/xapian. Accessed: 2025-08-19.

[68] Li Xu, Yili Hong, Max D Morris, and Kirk W Cameron. 2024. Prediction for distributional outcomes in high-performance computing input/output variability. *Journal of the Royal Statistical Society Series C: Applied Statistics* 73, 3 (2024), 561–580.

[69] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the Symposium on Cloud Computing (SOCC)*. ACM, 452–465. doi:10.1145/3127479.3131614

[70] Xin You, Zhibo Xuan, Hailong Yang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2024. GVARP: Detecting performance variance on large-scale heterogeneous systems. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[71] Jidong Zhai, Liyan Zheng, Feng Zhang, Xiongchao Tang, Haojie Wang, Teng Yu, Yuyang Jin, Shuaiwen Leon Song, and Wenguang Chen. 2022. Detecting performance variance for parallel applications without source code. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4239–4255.

[72] Yuxuan Zhao, Dmitry Duplyakin, Robert Ricci, and Alexandru Uta. 2021. Cloud performance variability prediction. In *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 35–40. doi:10.1145/34447545.3451182

[73] Liyan Zheng, Jidong Zhai, Xiongchao Tang, Haojie Wang, Teng Yu, Yuyang Jin, Shuaiwen Leon Song, and Wenguang Chen. 2022. Vapro: Performance variance detection and diagnosis for production-run parallel applications. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 150–162.

Appendix 1: VGO Workflow

VGO, which is built on SHARP [46], includes both a command-line interface (CLI) and a web-based graphical user interface (GUI) This appendix describes and depicts the GUI-based workflow of optimizing an application with VGO from start to finish. It is not required for understanding this paper, but it can shed light on how VGO can facilitate and automate an otherwise complex process. We will follow the methodology described in Section 3, but with additional steps, details, and screenshots of the workflow.

1. Measuring baseline distribution

SHARP was designed for reproducible performance evaluation, and therefore includes numerous controls and utilities to measure performance reliably when using VGO. Some of these options are visible in the GUI's "Measure" tab (Figure 12). After choosing a unique experiment (directory) name and an optional task (file) name, the user selects when to stop measuring. In the example shown, it is after exactly 1,000 repetitions, but SHARP includes a library of both static and adaptive stopping rules to determine when the distribution is statistically stable. Other inputs include where to run the experiment (backends), what other configuration files with metrics to include, and various other performance controls.

SHARP then runs the experiment and records all the pertinent information of an experiment in two human-readable and machine-readable files, which are shown to the user when the run is complete. The first, a CSV file, contains one row per program execution and one column per metric collected. The second, a markdown file, contains a detailed description of the system-under-test (SUT) and experiment parameters for reproducibility purposes. The user can also load the data in the "Explore" tab (Figure 13) to visualize the distribution as a density plot and summary statistics. In addition to a density plot, the graph shows a scatter plot of the individual runs, a boxplot with confidence intervals, and a Bayesian highest-density interval. The user can choose the main metric to visualize, filter the data to a particular subrange (for example, by time or by performance), and perform pairwise comparisons of different metrics to discover interesting correlations.

2. Collecting system metrics. After measuring the baseline performance distribution, the rest of the VGO workflow is continued in the "Profile" tab. To initiate it, the user is asked to select the metadata (markdown) file from the step that was just completed or from a previous measurement (Figure 14).

VGO reads the metadata and reruns the same experiment exactly, with all the previous parameters, except with the addition of one or more metric layers. These layers are defined as external YAML files (see example in Figure 15), so new layers of metrics are easy to add. Backend options for program execution are composable, like mathematical functions, and metric collection options are additive, allowing an arbitrary number of layers. In practice, since these layers also add overhead, they could skew the performance distribution, so it is advisable to use only one or two layers at a time. It is worth noting that if VGO finds logs for a previous profiling run, it offers the user the option to reuse these results, as they can be quite slow to generate.

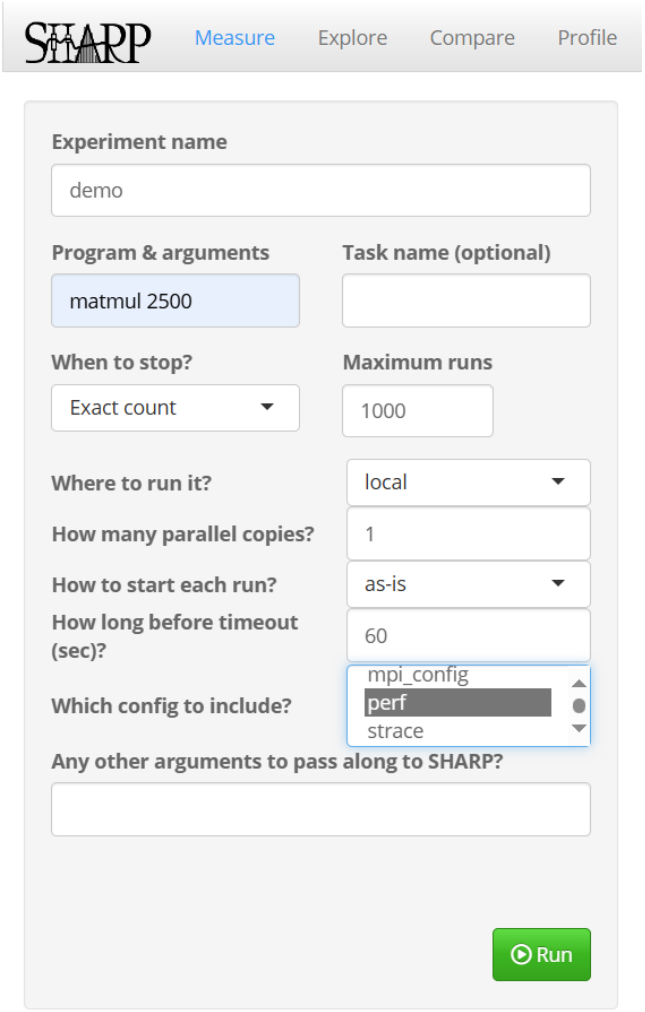


Figure 12: Example of launching a benchmark 1,000 times to generate performance distribution.

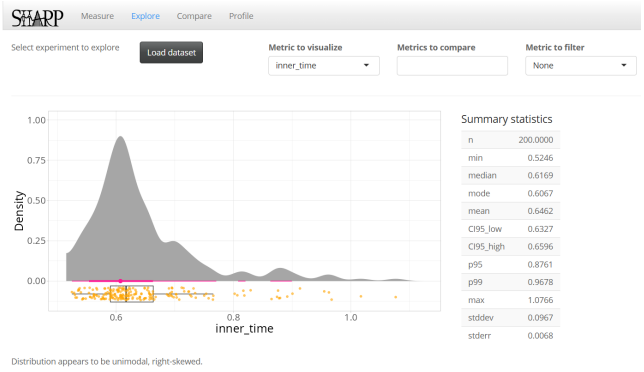


Figure 13: Example of distribution visualization.

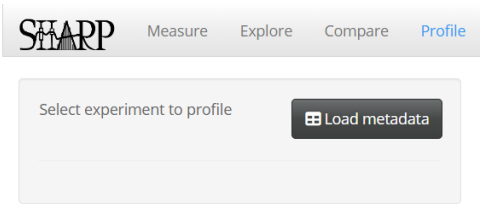


Figure 14: Initial view of the Profile tab

```
# Configuration excerpt to collect the number of major page faults
backend_options:
  bintime:
    run: /usr/bin/time -f "Major page faults: %R" $CMD $ARGS 2>&1
metrics:
  major_pagefaults:
    description: Total number of major page faults
    extract: awk '/Major page faults:/ {print $4}'
    lower_is_better: true
    type: numeric
    units: count
```

Figure 15: Example metric layer, including "how to measure" (*backend_options*) and "how to collect" (*metrics*).

3. Labeling performance

The next step is to inspect the distribution of any performance metric of interest, whether from the application or the metric layers, and identify properties of interest. First, the user selects the metric of interest ("Outcome metric") to see its distribution (Figure 16). Then, the user clicks anywhere on the distribution to place a cutoff line (dotted blue) that separates performance into two classes: "'LEFT" (green) and "RIGHT" (orange).

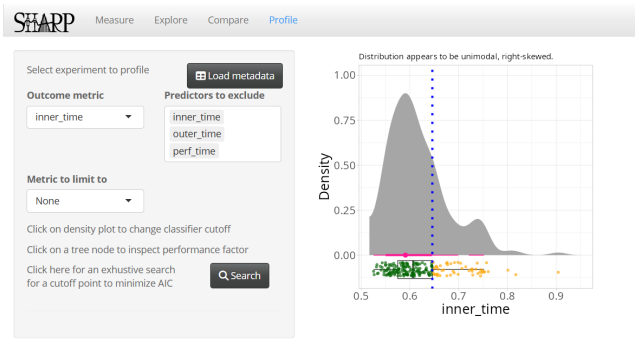


Figure 16: User interface to label performance classes.

In this example, the user chose a point somewhat right of the main mode to separate "slow performance" from "common performance", but could have also placed the line between the modes or even further right to isolate tail performance only. The user could also click on search to automatically find a good candidate for a cutoff point, as described later. Each such cutoff choice could lead to different results in subsequent steps, so it is worth experimenting with. Also note that the user may select to filter the samples inspected by any of the metrics ("Metric to limit to"), for example, to exclude invalid samples or focus on particular conditions.

4. Fitting a decision-tree classifier

For each selected cutoff line, VGO immediately fits a simple decision-tree classifier that attempts to predict which of the two performance classes a sample will fall into based on the values of its low-level system metrics. Since some low-level metrics are very strongly correlated with the outcome metric, the user interface lets the user select any number of metrics to exclude as potential predictors (Figure 16). VGO graphically shows the decision tree to the right of the distribution, which lets the user see which system factors affect performance variability. For example, in the tree shown in Figure 17, three low-level metrics collected from Linux's 'perf' tool are used to predict the "common" and "slow" performance classes. In particular, all of the "slow" (orange) measurements were associated with a high number of dTLB misses, while the majority of the measurements in the common case had fewer misses. This depiction makes it easy to see the important role that these misses play when performance varies away from the common case.

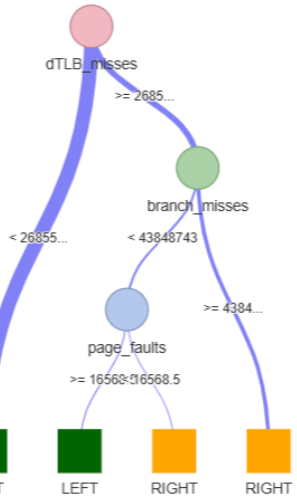
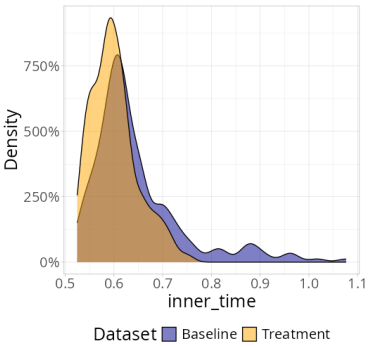


Figure 17: User interface for decision tree classifier.

5. Factor selection

The user may or may not be familiar with each of these factors, but VGO tries to help. Clicking on any of the tree nodes expands the view with a textual explanation of the selected factor, complete with URLs for additional references (Figure 18). These explanations are coded into another configuration file in YAML format, so the user can edit or add to these explanations, especially if they configure their metric layers beyond those provided. If all else fails, the user also has the option to click a button to prompt an optional, locally installed large-language model to explain this factor and its potential mitigation.

In addition to the textual explanation, VGO computes the R^2 relationship and draws a scatter plot of the selected outcome metric against the selected system factor, which in this example shows a strong linear relationship.



Statistic	Baseline	Treatment
n	200.0000	200.0000
min	0.5246	0.5247
median	0.6169	0.5939
mode	0.6067	0.5935
mean	0.6462	0.6010
CI95_low	0.6327	0.5944
CI95_high	0.6596	0.6076
p95	0.8761	0.6946
p99	0.9678	0.7237
max	1.0766	0.7550
stddev	0.0967	0.0474
stderr	0.0068	0.0034

Figure 20: Comparing baseline distribution to the mitigated one.

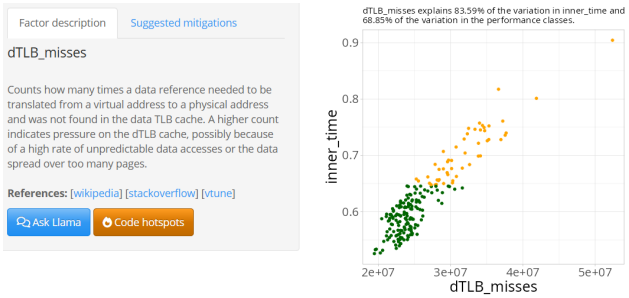


Figure 18: Visualizing and explaining the selected factor.

6. Mitigation selection

Having selected a promising factor, the user can next click on "Suggested mitigations" to get a drop-down list of potential actions to reduce the variability of this factor (Figure 19). Again, the user will see an explanation with references to the selected mitigation. And again, these are encoded in a separate YAML file that is easy to augment. Moreover, the YAML file can include instructions on how to programmatically apply the mitigation, when possible (a "mini Bash script"). Thus, when the user clicks on "Try it", VGO will be able to apply the mitigation on its own before remeasuring the distribution, which brings us to the next step. A list of common factors and mitigations is shown in Table 1.

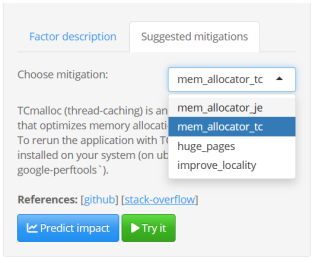


Figure 19: Potential mitigations list.

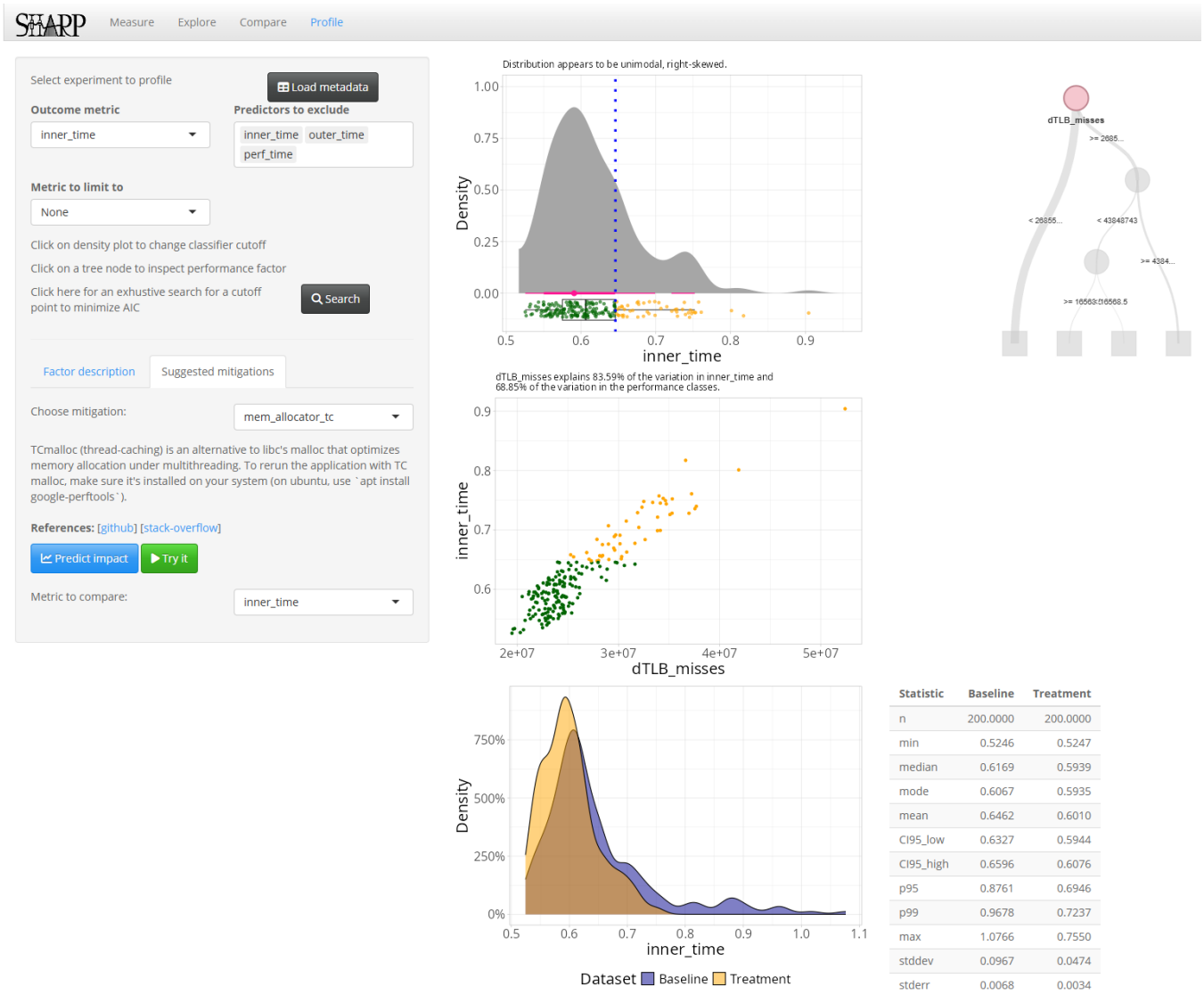
7. Remeasure distribution

Having selected a mitigation to try, VGO will either apply the mitigation itself when the instructions to do so are included in the YAML file or ask the user to do it manually before continuing. VGO then recreates the baseline experiment exactly, that is, with no profiling layers, but with the mitigation applied. Needless to say, if data from a previous application of the mitigation exists, the user has the option to skip rerunning it.

8. Distribution comparison

Finally, when the data from these runs is acquired, VGO compares the performance distributions with and without the mitigation (baseline). VGO depicts both a graphical summary of the two distributions overlaid on each other and a list of statistical summaries such as the mean, median, modes, and various dispersion measures of both distributions (Figure 20). At this point, the user can decide whether the newly reshaped distribution meets her goals, and accept the mitigation, or conversely, try a different mitigation or a different factor altogether. In that case, the user can simply choose from the same screen a new cutoff point, a new tree node, or a new mitigation, and try again until the performance distribution is satisfactory or the available choices are exhausted.

In summary, Figure 21 shows a screenshot of the final page view, encompassing nearly the complete workflow described here.



Statistic	Baseline	Treatment
n	200.0000	200.0000
min	0.5246	0.5247
median	0.6169	0.5939
mode	0.6067	0.5935
mean	0.6462	0.6010
CI95_low	0.6327	0.5944
CI95_high	0.6596	0.6076
p95	0.8761	0.6946
p99	0.9678	0.7237
max	1.0766	0.7550
stddev	0.0967	0.0474
stderr	0.0068	0.0034

Figure 21: The final view after completing the VGO workflow