benchkit: A Declarative Framework for Composable Performance Evaluation of System Software

Antonio Paolillo*
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
antonio.paolillo@vub.be

Mats Van Molle
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
mats.van.molle@vub.be

Ken Hasselmann
Department of Mechanics
Royal Military Academy
Brussels, Belgium
ken.hasselmann@mil.be

Abstract

Performance-evaluation pipelines in systems research often combine benchmarks, system configuration steps, profiling tools, and analysis scripts. In practice, these components are glued together with ad-hoc shell scripts, notebooks, and bespoke tooling, making experiment dimensions difficult to explore systematically and results hard to reproduce or extend.

We present `benchkit`, a lightweight Python library that provides a structured way to express performance experiments declaratively and to automate their full lifecycle—from build and execution to system configuration, profiling, and result collection. Instead of relying on monolithic scripts, `benchkit` provides a structured way to compose existing system tools (e.g., CPU-placement utilities, frequency controllers, and performance profilers) while keeping benchmark code untouched.

We illustrate `benchkit` through two representative studies: (1) a drilldown of performance anomalies in SPEC CPU workloads on hybrid-core x86 processors, enabled by systematic exploration of CPU placement policies; and (2) an analysis of lock implementations and scheduling strategies on a many-core ARM server, where `benchkit` coordinates system tools and visualizations to interpret performance differences. We evaluate the overhead of `benchkit` and show that it introduces no measurable cost compared to hand-written shell workflows, both on the host and inside containers. These results show that `benchkit` provides a reproducible, extensible, and principled foundation for system-level performance experimentation.

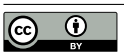
CCS Concepts

• **Software and its engineering** → **Software performance**; *Empirical software validation*; • **General and reference** → *Performance*.

Keywords

performance, benchmarking, reproducibility, composability, experiment automation, profiling, scalability

*Corresponding author.



ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3796997

ACM Reference Format:
Antonio Paolillo, Mats Van Molle, and Ken Hasselmann. 2026. `benchkit`: A Declarative Framework for Composable Performance Evaluation of System Software. 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, 14 pages. https://doi.org/10.1145/3777884.3796997

1 Introduction

Performance evaluation of software systems routinely involves combining benchmarks, system-level configuration (e.g., thread placement, frequency settings, NUMA policies), profiling tools such as `perf`, and post-processing pipelines including flame graph generation. In practice, these components are commonly assembled through ad-hoc shell scripts, undocumented environment variables, and project-specific workflows. Such setups are fragile, difficult to extend, and hard to reproduce—in particular when experiments involve multiple tools, parameter spaces, or platform-dependent transformations.

Despite their diversity, most benchmarking workflows follow the same structure: **(1) Fetch:** obtain the benchmark sources or binaries and prepare the environment; **(2) Build:** configure and compile the benchmark with specific build-time parameters; **(3) Run:** execute the benchmark program under a chosen configuration; and **(4) Collect:** gather performance metrics or artifacts for analysis. Before or during execution, external tools are often used to modify behavior (e.g., `taskset`, `numactl`, `cpupower`, `LD_PRELOAD`) or to record additional information (`perf-stat`, `perf-record`, `trace-cmd`, `strace`, `lscpu`). Reproducing such multi-layered pipelines requires not only re-running the benchmark, but also re-applying environment transformations in the correct order with consistent parameters. Manually orchestrating these layers increases the risk of configuration drift, undocumented assumptions, and inconsistent analysis. Existing tools provide fragments of this workflow, but no coherent, reusable, or extensible way to express complete performance experiments.

`benchkit` addresses these challenges by providing a lightweight Python library that offers a structured way to express performance experiments declaratively. It exposes *campaigns*, *parameter spaces*, *wrappers*, *hooks*, and *platform abstractions* as first-class objects, enabling users to compose system-level transformations and measurement tools using existing command-line utilities without modifying benchmark code. By representing the complete fetch–build–run–collect workflow as executable code, `benchkit` ensures transparency, repeatability, and portability across platforms and configurations.

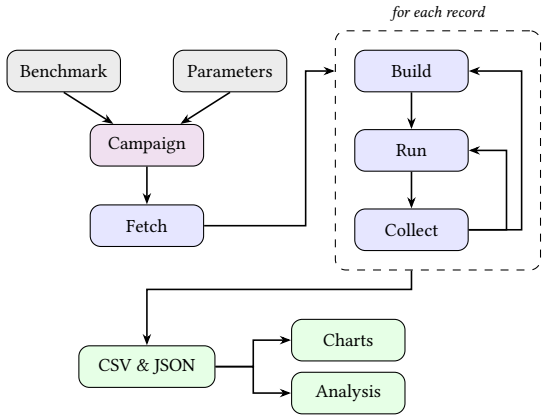


Figure 1: Overview of the benchkit workflow. A campaign combines a benchmark with a parameter space and iterates over all configurations, producing structured CSV and JSON results, charts, and analysis.

A central challenge in performance evaluation is *reproducibility* [7, 44]. Small changes in CPU scheduling, frequency scaling, or OS interference can lead to significant variation across runs. Modern hybrid performance/efficiency-core processors exacerbate this issue: even single-threaded workloads may exhibit large performance disparities depending on the core they execute on. benchkit provides the necessary structure to detect, control, and analyze such effects by enabling systematic exploration of system-level settings such as thread placement and frequency control, and by integrating profiling and visualization tools (e.g., `perf-stat`, `perf-record`, flame graphs) within a unified experiment definition.

Problem Statement. Although many individual tools exist for profiling, tracing, and benchmarking, there is no coherent and reusable way to combine them into reliable, self-documenting performance experiments. Current practice relies on ad-hoc scripts that are hard to maintain, difficult to extend, and unsuitable for systematic exploration of parameter spaces. What is needed is a library that: (i) expresses benchmarks and their variants in a modular structure, (ii) automates the integration of external system tools, and (iii) produces transparent, repeatable, and reusable experimental artifacts. benchkit is designed to provide this missing structure and unify the end-to-end performance evaluation workflow.

Contributions. We contribute benchkit, a lightweight and extensible library for systematic performance experimentation. Its capabilities and usefulness are demonstrated through:

- a *performance drilldown* using a representative SPEC CPU workload on a hybrid-core x86 processor, where benchkit detects large cross-run performance variations, explores CPU placement systematically, and integrates profiling tools to support root-cause analysis;
- an *illustrative system study* on a many-core ARM server, showing how benchkit composes locking mechanisms with scheduling policies, and provides visual analyses to interpret performance differences; and
- an *overhead evaluation* demonstrating that benchkit introduces no measurable cost compared to equivalent hand-written shell workflows, both on host systems and inside containers.

These case studies highlight how benchkit provides a reproducible, extensible, and principled foundation for systematic system-level performance experimentation.

The remainder of this paper is structured as follows. Section 2 presents an overview of benchkit and its core abstractions, including wrappers, hooks, and platforms. Section 3 introduces our experimental methodology and frames the drilldown investigation conducted on representative SPEC CPU workloads. Section 4 presents our study of locks and scheduling policies on a large ARM server. Section 5 evaluates benchkit's overhead. Section 6 discusses related work. Section 7 discusses limitations. Section 8 concludes.

2 Framework Overview

2.1 Core Abstractions

benchkit provides three fundamental abstractions—*benchmarks*, *campaigns*, and *parameter spaces*—which together define the structure of a performance experiment.

Benchmarks. A *benchmark* encapsulates all information required to obtain, build, and execute a workload program, and to collect its results. Users implement four methods: **(1) `fetch()`:** obtain the benchmark sources or binaries (e.g., clone a git repository, extract an archive, or copy from a local path); **(2) `build()`:** configure and compile the workload using build-time parameters (e.g., by invoking CMake and/or a Makefile); **(3) `run()`:** execute the workload with run-time parameters; and **(4) `collect()`:** extract metrics from the workload's output or from external tools invoked during the run. Both build-time and run-time parameters are part of the benchmark definition, enabling fine-grained control over configuration.

Parameter Spaces. The *parameter space* defines the variables explored during a campaign (e.g., CPU core selection, frequency settings, thread counts, allocator types). benchkit supports both cartesian-product iteration—i.e., evaluating all possible combinations of variables—and custom-defined exploration strategies (e.g., sampling subsets, sweeping only selected axes). Each concrete assignment of values to all variables is called a *record*; the list of records to sweep is derived from the given parameter space. Variable names correspond directly to the build-time and run-time parameters consumed by the benchmark methods introduced above.

Campaigns. A *campaign* specifies how to explore the parameter space and orchestrates the full fetch–build–run–collect workflow for each configuration. Campaigns compose benchmarks with parameter spaces and optional instrumentation (wrappers, hooks, shared libraries introduced below), producing self-contained and reproducible experiment definitions. Together, these abstractions compose cleanly and prevent ad-hoc, project-specific workflows.

2.2 Execution Workflow

Figure 1 summarizes the execution workflow. Users instantiate a campaign by providing a benchmark, a parameter space, and optional wrappers or hooks that modify behavior or collect data.

Algorithm 1: benchkit campaign execution workflow. The **platform** abstracts the target environment (local, Docker, SSH, environment variables, etc.); the **run** method uses it to spawn a process with the benchmark command and capture its standard output.

Input: records – List of parameter records
benchmark – Benchmark to evaluate
platform – Target execution platform
Output: results – List of result mappings (names to values)

```
results ← empty list;
(fetch_params, records) ← split_fetch(records);
benchmark.fetch(platform, fetch_params);
foreach record in records do
    (build_params, run_params) ← split(record);
    foreach bp in build_params do
        benchmark.build(platform, bp);
        foreach rp in run_params do
            output ← benchmark.run(platform, rp);
            result ← benchmark.collect(output);
            rr ← bp ∪ rp ∪ result;
            results.append(rr);
return results;
```

Algorithm 1 illustrates the core execution flow: records from the parameter space are split into build-time and run-time components; the benchmark is rebuilt when necessary; and each run produces a structured result that merges inputs and outputs.

benchkit automatically aggregates results into one CSV file (one row per parameter combination) and a JSON file per run. Support for repeated runs (nb_runs in the campaign definition) enables statistical analysis and mitigates the impact of natural variability.

2.3 Example

Listing 1 shows a concise benchkit campaign evaluating two microbenchmarks provided in the LevelDB [22] codebase across different thread counts. For clarity, the example omits some boilerplate (e.g., full fetch() logic), but the structure mirrors how benchkit runs real campaigns.

The user defines a benchmark class, a parameter space, and a campaign object. benchkit handles all orchestration and produces both CSV results and visualizations.

This example highlights the declarative nature of benchkit: the benchmark and its parameter space are defined in Python, while benchkit handles orchestration, data collection, and storage.

2.4 Meta-Benchmarking Abstractions

Beyond the core abstractions, benchkit exposes mechanisms that allow users to control the execution environment, integrate additional analysis tools, and produce structured artifacts. These capabilities form the "meta-benchmarking" layer that models how system tools interact with a benchmarked program.

Hooks. Hooks execute user-defined logic before and after each run. *Pre-run hooks* can configure system state, launch daemons, or adjust parameters dynamically; *post-run hooks* can collect profiling data (e.g., perf record) and append them to results. All returned information is automatically merged into the output record.

```python
from benchkit import CampaignCartesianProduct

class LevelDBBench:
  # Simplified for illustration
  def fetch(self, platform):
    url = "https://github.com/google/leveldb.git"
    platform.shell(f"git clone {url} src")

  def build(self, platform):
    b = "src/build"
    platform.makedir(b)
    platform.shell("cmake ..", dir=b)
    platform.shell("make", dir=b)

  def run(self, platform, bench_name, nb_threads):
    out = platform.run_shell(
      f"./db_bench"
      f" --benchmarks={bench_name} --threads={nb_threads}")
    return out

  def collect(self, output):
    count = int(parse_count(output))
    dur   = float(parse_duration(output))
    return {"throughput": count / dur}

parameter_space = {
  "bench_name": ["readrandom", "seekrandom"],
  "nb_threads": [2, 4, 8],
}

campaign = CampaignCartesianProduct(
  benchmark=LevelDBBench(),
  variables=parameter_space,
)
campaign.run()
campaign.generate_graph(
    plot_name="lineplot",
    x="nb_threads",
    y="throughput",
    hue="bench_name",
)
```

Listing 1: Example of a benchkit campaign for LevelDB.

Each run receives a dedicated directory in a hierarchy that mirrors the parameter space, ensuring that logs, artifacts, and profiling outputs are isolated and attributable to a specific configuration. For instance, a campaign sweeping bench_name and nb_threads (as in Listing 1) produces:

```
bench_name-readrandom/nb_threads-2/run-01/{results.json,...}
bench_name-readrandom/nb_threads-4/run-01/{results.json,...}
bench_name-seekrandom/nb_threads-2/run-01/{results.json,...}
bench_name-seekrandom/nb_threads-4/run-01/{results.json,...}
```

Command Wrappers. Wrappers compose external tools around the benchmark command. They cleanly decouple profiling or instrumentation from the benchmark implementation. Examples include perf-stat, perf-record, strace, trace-cmd, or modifiers such as env and taskset. Tools such as cpupower are typically invoked from hooks instead.

Shared Libraries. Through LD_PRELOAD, benchkit can inject custom shared libraries (e.g., tcmalloc [21], jemalloc [16], or synchronization libraries such as LiTL [31] or tilt, introduced below) without modifying the benchmark code. This is particularly useful for experiments involving memory allocators, locking strategies, or interposition-based tooling. In practice, benchkit automatically

sets the required environment variables (e.g., `LD_PRELOAD` paths) in the platform before executing the benchmark, so the user only declares which libraries to use.

Platform Abstraction. A platform is a first-class object that encapsulates how commands are executed on the benchmarked machine, cleanly separating orchestration (from the host running `benchkit`) from execution (on the target platform). `benchkit` currently supports platform models for: (i) local execution on Linux (via `get_current_platform()`), (ii) containerized environments (Docker, with native `pythainer` [49] integration), (iii) remote nodes (via SSH), and (iv) mobile or embedded targets (Android via ADB; OpenHarmony via HDC). Campaigns can therefore run unmodified across multiple environments.

Integration. Algorithm 2 refines the core workflow of Algorithm 1 by showing where these meta-benchmarking mechanisms integrate into the execution loop. Shared libraries are fetched once and rebuilt when build-time parameters change, then injected by setting the appropriate environment variables (e.g., `LD_PRELOAD`) in the platform; pre-run hooks prepare the execution context before each run; command wrappers compose around the benchmark command via the platform abstraction; and post-run hooks tear down the execution context and augment results with additional metrics.

2.5 Repeatability and Reproducibility

We adopt the ACM artifact-review terminology [1], which distinguishes three levels of result validation. **Repeatability** (same team, same setup) is ensured because all parameters, hooks, wrappers, and artifacts are encoded in a single campaign definition that can be re-executed identically. **Reproducibility** (different team, same setup) is facilitated because the campaign definition, together with the benchmark and wrapper implementations, constitutes a self-contained artifact that an independent group can execute on the same platform. **Replicability** (different team, different setup) is supported by `benchkit`'s platform abstraction, which allows a campaign to be ported across architectures with minimal or no modification to the experiment definition.

Run-to-run variability, caused by OS noise, scheduling interference, and hardware effects, remains an inherent challenge. However, `benchkit` mitigates these effects by: (i) optionally fixing CPU frequencies, (ii) disabling noise-inducing services (e.g., `irqbalance`) through *hooks*, (iii) supporting repeated runs, and (iv) structuring all experiment metadata for inspection. We note, however, that full reproducibility and replicability of system-level experiments are inherently limited by factors outside the framework's control: hardware performance counters, kernel scheduler behavior, and low-level system interfaces may differ across machines and kernel versions, and containerization does not isolate the host kernel. `benchkit` cannot abstract away these dependencies, but its structured campaign definitions and extensive output capabilities—which automatically document hardware, OS, kernel version, CPU frequency governors, and other system state at experiment time—make them explicit, aiding diagnosis when results diverge.

Algorithm 2: `benchkit` extended workflow with meta-benchmarking. Colored lines show where shared libraries, pre-run hooks, command wrappers, and post-run hooks integrate into Algorithm 1.

Input: records, **benchmark**, **platform**,
shared_libs, **pre_hooks**, **wrappers**, **post_hooks**
Output: results – List of result mappings (names to values)

```
results ← empty list;
(fetch_params, records) ← split_fetch(records);
benchmark.fetch(platform, fetch_params);
foreach lib in shared_libs do
    lib.fetch();
foreach record in records do
    (build_params, run_params) ← split(record);
    foreach bp in build_params do
        benchmark.build(platform, bp);
        foreach lib in shared_libs do
            lib.build(bp);
        foreach rp in run_params do
            env ← preload_env(shared_libs, bp, rp);
            foreach hook in pre_hooks do
                hook(bp, rp);
            platform ← wrap(wrappers, platform, env, bp, rp);
            output ← benchmark.run(platform, rp);
            result ← benchmark.collect(output);
            foreach hook in post_hooks do
                result ← result ∪ hook(result);
            rr ← bp ∪ rp ∪ result;
            results.append(rr);
return results;
```

2.6 Open Source

`benchkit` has been under active development since 2021 and was publicly released in 2024. It includes a growing catalog of ready-to-run workloads, as defined by the benchmark abstraction of Listing 1. These cover several domains: databases (LevelDB [22], RocksDB [17], Kyoto Cabinet [20], Redis [40], Memcached [15], MySQL [48], PostgreSQL [58] with sysbench [35]), standard CPU benchmark suites (SPEC CPU 2017 [5], with automated installation from ISO), microbenchmarks (membench [56], will-it-scale [4], locktorture [42]), large-scale workloads (MapReduce [11], NAS Parallel Benchmarks [3]), real-time workloads (`cyclictest` [57]), and full support for the Phoronix Test Suite [52], including parsing, execution, and result collection according to the Phoronix benchmark specification. This catalog enables rapid exploration of new configurations and straightforward replication of existing studies.

In addition to benchmarks, `benchkit` ships with ready-to-use command wrappers (`perf-stat`, `perf-record`, `strace`, `ltrace`, `numactl`, `taskset`, `valgrind`, `trace-cmd`, NVIDIA Nsight Systems and Nsight Compute), shared-library integrations (prebuilt libraries, `tilt`), and asynchronous command attachments for BPF-based tracing (`klockstat`, `offcputime`, `llcstat`).

```python
from benchkit import CampaignCartesianProduct

class SpecBench:
  [...] # Definition similar to LevelDB

campaign = CampaignCartesianProduct(
  benchmark=SpecBench(path="/path/to/cpu2017.iso"),
  variables={
    "bench_name": ["500.perlbench"],
  },
  nb_runs=10, # Repeated for variability check
)

campaign.run()
```

Listing 2: Example of a `benchkit` campaign for a SPEC single-threaded benchmark.

```python
import os
from benchkit.benches.heater import heater_seq_campaign

campaign = heater_seq_campaign(
    nb_runs=3,
    duration_s=3,
    cpu=range(0, os.cpu_count()),
)
campaign.run()
campaign.generate_graph(
    plot_name="barplot", x="cpu", y="ops",
    title="Sequential heater - per-CPU throughput",
)
```

Listing 3: `benchkit` campaign sweeping all available CPUs for the sequential heater benchmark.

`benchkit` is available as an open-source project under the MIT license.[1] The experiments and code listings presented in this paper are available as a replication package.[2]

3 Drilldown Case Study on Hybrid–Core Variability

Modern hybrid-core processors mix high-performance (P) cores with energy-efficient (E) cores. While this architecture improves power efficiency, it complicates performance evaluation: even single-threaded programs may experience large run-to-run variability depending on the core on which they begin execution. This variability is pronounced under default Linux scheduling, where processes may migrate across heterogeneous cores.

This section demonstrates how `benchkit` enables a systematic investigation of such effects using a representative single-threaded SPEC CPU workload. We first measure its performance under default scheduling and observe significant variability. We then characterize the underlying platform through a sequential CPU-ID heatmap, revealing performance differences between P and E cores. Finally, we reproduce the experiment under controlled placements using `taskset`-based pinning, collapsing the variability and confirming the root cause.

3.1 Benchmark Setup and Baseline Experiment

Our experiments use the `500.perlbench_r` workload from the SPEC CPU 2017 suite, executed on a modern hybrid-core x86 laptop equipped with an AMD Ryzen AI 9 HX 370 processor (12 cores, 24 threads, 32 GiB RAM) running Linux 6.17.

Listing 2 shows a simplified benchmark definition in `benchkit`. The campaign repeatedly executes the workload under the default Linux scheduler to expose run-to-run variability. The parameter space contains only the workload name; repeated executions are controlled through the campaign's `nb_runs` field.

`benchkit` automatically generates one directory per run, stores the benchmark's standard output and all artifacts inside it, and produces a CSV summary aggregating all runtimes. Users supply the SPEC CPU archive via the `path` argument; `benchkit` then handles unpacking, building, and executing the workload according to the declared parameters, without requiring any modifications to the benchmark itself.

Figure 2 reports the runtimes of `500.perlbench_r` across ten repeated executions under the default scheduler, with no affinity or placement control (see left figure, blue dots). The resulting measurements show substantial variability: runtimes range from 119.6 s to 148.9 s, a spread exceeding 24 % relative to the minimum, with a standard deviation of 10.1 s—unusually large for a deterministic single-threaded workload.

This multimodality is characteristic of hybrid-core architectures: the workload may begin on either a fast P-core or a slower E-core, and may be migrated between them during execution. Both effects are visible in Figure 2 (left, blue dots), where several intermediate runtimes correspond to runs that began on one class of core and were migrated to the other.

This baseline experiment highlights the necessity of explicit thread placement when evaluating performance on heterogeneous processors and motivates the next steps of our drilldown analysis.

3.2 Sequential CPU–ID Characterization

To understand the source of the observed variability, we next characterize the relative speed of each logical CPU on the hybrid processor. For this purpose, we implement a simple *sequential heater* benchmark that runs a fixed amount of integer work while being pinned to a single core.

The heater is a small C program (Listing omitted for brevity) that: (i) parses a target duration and CPU identifier from the command line, (ii) uses `sched_setaffinity` to pin itself to the requested core, and (iii) performs a tight arithmetic loop until the duration has elapsed, counting the number of operations performed and printing the final count (as `Operations performed: <N>`). The work per loop iteration is fixed, so the resulting operation count is proportional to the effective compute capacity of the selected core.

We expose this program to `benchkit` through a dedicated benchmark object `HeaterSeqBench`, whose parameter space contains a single variable `cpu` representing the target core. To systematically explore placement effects, `benchkit` provides a helper function, `heater_seq_campaign()`, which constructs a campaign sweeping all available CPUs. Listing 3 shows the code used to instantiate and run this experiment, together with the command generating a barplot of per-core throughput.

[1] https://github.com/open-s4c/benchkit
[2] https://github.com/softwarelanguageslab/icpe26-benchkit-ae

```python
from benchkit import CampaignCartesianProduct
from benchkit.benches import SpecBench
from benchkit.commandwrappers.taskset import TasksetWrap

P_CORES = [0, 1, 2, 3, 12, 13, 14, 15]
E_CORES = [4, 5, 6, 7, 8, 9, 10, 11,
           16, 17, 18, 19, 20, 21, 22, 23]

campaign = CampaignCartesianProduct(
    benchmark=SpecBench(path="/path/to/cpu2017.iso"),
    command_wrappers=[TasksetWrap()],
    variables={
        "bench_name": ["500.perlbench"],
        "cpu_list": {
            "no_pinning": [],
            "p_cores":    P_CORES,
            "e_cores":    E_CORES,
        },
    },
    pretty={"cpu_list": {
        "no_pinning": "No Pinning",
        "p_cores":    "P-Cores",
        "e_cores":    "E-Cores",
    }},
    nb_runs=10,
)

campaign.run()
campaign.generate_graph(
    plot_name="stripplot", x="cpu_list", y="duration_s",
)
```

Listing 4: `benchkit` campaign using `TasksetWrap` to enforce core placement for `500.perlbench_r`.

Figure 2 (right) shows the resulting operation counts for each CPU on our platform. Two clear plateaus emerge: cores 0–3 and 12–15 achieve around 1.27×10^9 operations, while cores 4–11 and 16–23 reach only about 0.81×10^9 operations. This pattern is consistent with the presence of two classes of cores (high-performance P-cores and energy-efficient E-cores), and it provides an explicit mapping from core identifiers to their relative speed.

This characterization step is fully automated by `benchkit`: the campaign definition expresses the CPU sweep declaratively, and the resulting per-core measurements and plot are generated without additional hand-written scripting. In the next subsection, we use this mapping to re-run the SPEC workload under controlled placements on P-cores and E-cores only.

3.3 Controlled Placement with `taskset`

Using the core-to-performance mapping obtained from the sequential heater (Section 3.2), we now rerun `500.perlbench_r` under three placement configurations: *No Pinning* (default scheduler, as in Listing 2), *P-Cores only*, and *E-Cores only*. On our platform, the heater identifies cores `0–3` and `12–15` as fast P-cores, and cores `4–11` and `16–23` as slower E-cores. These ranges form the basis for the corresponding `taskset` placements used below.

In `benchkit`, these placement policies are expressed declaratively using the `TasksetWrap` command wrapper. The wrapper receives, for each run, the value of the `cpu_list` variable drawn from the parameter space; `benchkit` then injects it automatically when wrapping the benchmark command (as described in Algorithm 2). This illustrates a key aspect of our model: variables in the parameter space are not only passed to benchmarks, but can also configure wrappers and hooks, enabling fine-grained control of system behavior. Listing 4 shows the corresponding campaign.

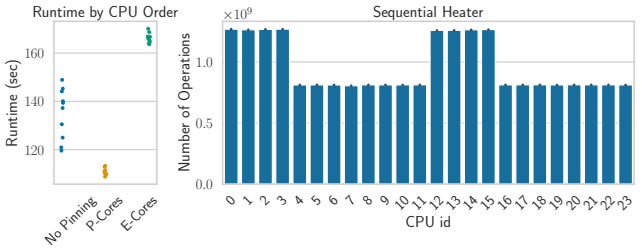


Figure 2: Runtimes of `500.perlbench_r` under three thread-placement configurations: default scheduling (*No Pinning*), pinned to P-cores, and pinned to E-cores. Each point represents a single run (left). Sequential heater results shown for the hybrid-core AMD Ryzen AI 9 HX 370, which features four P-cores and eight E-cores (with SMT: logical CPUs 0–3, 12–15 and 4–11, 16–23) (right).

Figure 2 (left) summarizes the resulting runtimes. The *No Pinning* configuration reproduces the wide spread observed in Section 3.1. When restricting execution to P-cores, runtimes cluster tightly around the minimum (≈110 s), indicating stable performance on the fast cores. Conversely, pinning to E-cores yields consistently slower runtimes (≈167 s), with little intra-group variation.

This experiment confirms that the multimodal behavior of the SPEC workload stems from uncontrolled placement on heterogeneous cores. It also illustrates how `benchkit` composes system-level tools such as `taskset` within a single, reusable experiment description, without modifying the (externally defined) benchmark or writing ad-hoc shell scripts.

4 Composing System Mechanisms with `benchkit`: Locks, Schedulers, Profilers

Beyond single-workload drilldowns, `benchkit` is designed to compose multiple system mechanisms—e.g., locking libraries, scheduling policies, and profiling tools—within a single coherent experiment. This section demonstrates how `benchkit` models such mechanisms using the same abstractions introduced earlier (wrappers, hooks, shared libraries), enabling reproducible studies that span locks, schedulers, and hardware-based profiling within a unified experimental sweep.

We integrate: (i) `tilt`, a drop-in library of lock implementations exposed via `pthread` interposition; (ii) `schedkit`, a user-space scheduler that adjusts thread affinities according to NUMA-aware policies; and (iii) the Linux `perf` suite for collecting hardware and kernel metrics. These components illustrate how `benchkit` treats system mechanisms as first-class composable elements rather than ad-hoc scripts layered around workloads. The resulting campaigns combine lock algorithms, scheduling strategies, and profiling instrumentation directly within the parameter space and configuration of a single `benchkit` experiment.

All experiments in this section were conducted on a NUMA-enabled Taishan 200 Kunpeng server with two Kunpeng 920–4826

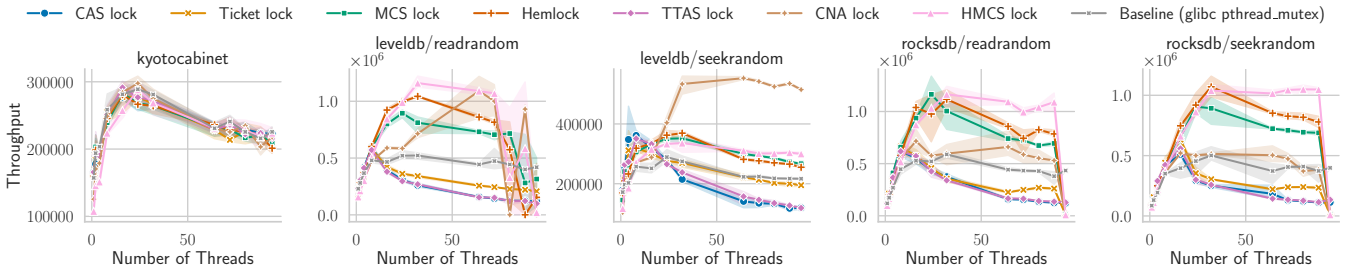


Figure 3: Throughput performance for locking experiments across Kyoto Cabinet, LevelDB, and RocksDB benchmarks as a function of the number of threads. NUMA-aware hierarchical locks outperform flat locks at high thread counts. Results shown for a 96-core Kunpeng 920 server (4 NUMA nodes).

processors (96 cores, 4 NUMA nodes, 539 GB RAM), running Ubuntu 20.04.6 LTS with kernel 5.4.0-200-generic (aarch64).

4.1 Studying Locking Impact

The choice of locking algorithm can significantly influence performance. benchkit makes it easy to vary the locking mechanism used during an experiment and to evaluate its impact systematically. tilt[3] is a shared library that provides a drop-in replacement for pthread synchronization calls (e.g., pthread_mutex_lock and related functions). By interposing on these calls, tilt enables the evaluation of different locking mechanisms, such as Ticket lock, MCS lock, and NUMA-aware hierarchical locks. tilt is inspired by LiTL [27, 31] but is designed to be lightweight, does not require per-lock dynamic memory allocation or auxiliary data structures (though it can support them), and is actively maintained. Its lock implementations are drawn from libvsync [47], a library of synchronization primitives verified and optimized for weak memory models as part of the VSync project [10, 45].

In this work, we use tilt to integrate seven spinlock algorithms from the literature. These include flat locks—such as CAS, TTAS [30], Ticket [38], MCS [43], and Hemlock [13]—as well as NUMA-aware hierarchical designs like CNA [12] and HMCS [6, 46]. As a baseline, we compare against the default glibc implementation of pthread_mutex, used without tilt.

Manually, a custom lock implementation based on tilt can be enabled by running a benchmark with LD_PRELOAD, for example: LD_PRELOAD=./libmutrep.so ./db_bench .... This instructs the ELF loader to load libmutrep.so before other shared libraries, allowing it to intercept pthread calls and replace them with the selected lock algorithm at run time.

In benchkit, tilt is modeled as a SharedLibrary component. This abstraction automatically configures LD_PRELOAD for each run, ensuring that the correct library variant is injected into the benchmark environment. Lock implementations then become ordinary run-time parameters in the benchkit campaign: a single benchmark can be evaluated under multiple locks by varying a lock variable in the parameter space, as the SharedLibrary receives the variable–value mapping corresponding to the current record.

[3]https://github.com/open-s4c/tilt

```python
from benchkit import CampaignCartesianProduct
from benchkit.benches.leveldb import LevelDBBench
from benchkit.sharedlibs.tiltlib import TiltLib

campaign = CampaignCartesianProduct(
    benchmark=LevelDBBench(),
    shared_libs=[TiltLib()],
    variables={
        "nb_threads": [1, 2, 4, 8, 16, 24, 32,
                       64, 72, 80, 88, 96],
        "bench_name": ["readrandom"],
        "lock": ["caslock", "ticketlock", "mcslock", "hemlock",
                 "ttaslock", "cnalock", "hmcslock", "pthread"],
    },
    nb_runs=3,
)
campaign.run()
```

Listing 5: benchkit campaign for LevelDB with different locks.

benchkit rebuilds the tilt library whenever its build-time parameters (e.g., selected lock, optimization flags) change, ensuring consistency across configurations and avoiding ad-hoc scripting.

Listing 5 illustrates a benchkit campaign evaluating different lock implementations for the *LevelDB: readrandom* workload. The same approach applies to other benchmarks (e.g., RocksDB, Kyoto Cabinet), and Figure 3 presents the corresponding performance results across this broader set of workloads.

4.2 Studying Thread–Placement Impact

Scheduling and thread placement are system-level mechanisms that can significantly influence benchmark performance, especially on large NUMA-enabled servers. To evaluate such mechanisms—in particular, placement strategies and scheduler policies under varying contention and locality conditions—in a controlled and reproducible manner, we use schedkit[4], a lightweight user-space scheduler that adjusts thread affinities during program execution.

schedkit is a Python framework designed as fully plugin-based: each scheduling policy is implemented as an independent module, allowing developers to extend or modify placement strategies without changing the core runtime. Policies may rely on platform characteristics such as CPU utilization, NUMA topology, cache-sharing

[4]https://github.com/open-s4c/schedkit

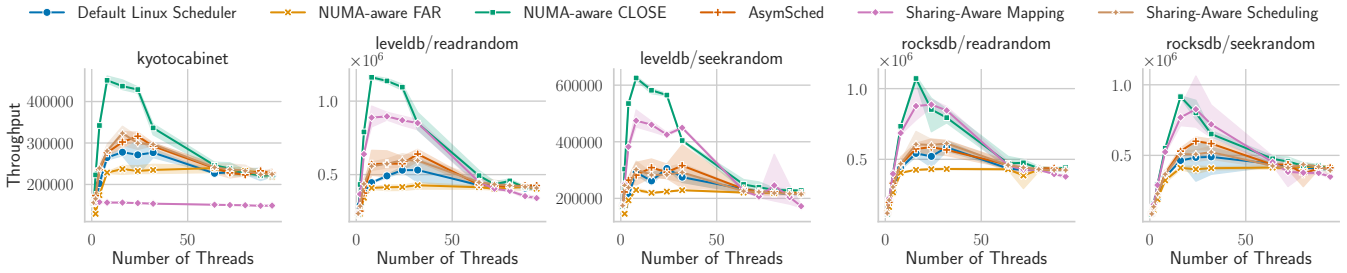


Figure 4: Throughput performance for scheduling experiments across Kyoto Cabinet, LevelDB, and RocksDB benchmarks as a function of the number of threads. NUMA-aware CLOSE scheduling consistently outperforms other policies by improving cache locality, while the FAR policy struggles due to increased memory access latencies.

```python
class SchedKitHooks:
  def __init__(self, platform):
    self.platform = platform

  def start_sched_hook(self, platform, variables):
    pol = variables["scheduler"]
    platform.comm.background_subprocess(
        command=["./schedkit", "--policy", pol],
    )

  def end_sched_hook(self, platform, variables):
    platform.shell(["killall", "-15",
                    "schedkit"])
```

Listing 6: Pre-run and post-run hooks for `schedkit`.

structure, or memory-access distances. On multi-socket NUMA systems, remote memory accesses incur higher latency [10, 39], making thread placement directly influence locality, communication patterns, and cache behavior. Accordingly, `schedkit` policies can either co-locate threads to improve locality (e.g., *CLOSE*) or disperse them to reduce cache interference (e.g., *FAR*). In addition, we integrated into `schedkit` three policies from prior work—AsymSched [37], SAM [54], and SAS [55]—to study them under identical experimental conditions.

Within `benchkit`, `schedkit` is modeled using `PreRunHook` and `PostRunHook` callbacks. Before each run, a `PreRunHook` launches the `schedkit` daemon in the background with the policy specified for the current run, ensuring that the placement strategy is active for the duration of the benchmark. After completion, a `PostRunHook` callback terminates the daemon so that no residual state persists across runs. Because the variables of the current record are passed to each hook callback, `benchkit` can treat scheduling policies as ordinary run-time parameters: a campaign simply varies a `scheduler` variable in the parameter space, and the hooks automatically receive and apply the selected value.

Listing 6 shows the minimal hook implementations used to start and stop `schedkit` around each run. Listing 7 shows how a `benchkit` campaign sweeps over multiple scheduling policies for the *LevelDB: readrandom* workload. The same approach applies to other workloads such as `RocksDB` and `Kyoto Cabinet`, and the results across all workloads are reported in Figure 4.

```python
from benchkit import CampaignCartesianProduct
from benchkit.benches.leveldb import LevelDBBench
from benchkit.platforms import get_current_platform

platform = get_current_platform()
schedkit = SchedKitHooks(platform=platform)

schedulers = ["Normal", "FAR", "CLOSE",
              "AsymSched", "SAM", "SAS"]

campaign = CampaignCartesianProduct(
    benchmark=LevelDBBench(),
    pre_run_hooks=[schedkit.start_sched_hook],
    post_run_hooks=[schedkit.end_sched_hook],
    variables={
        "nb_threads": [1, 2, 4, 8, 16, 24, 32,
                       64, 72, 80, 88, 96],
        "bench_name": ["readrandom"],
        "scheduler": schedulers,
    },
    nb_runs=3,
)
campaign.run()
```

Listing 7: `benchkit` campaign for LevelDB with different scheduling policies.

4.3 Using `perf` for Profiling and Run-Time Statistics

In many experiments, raw benchmark results (e.g., throughput or latency) are insufficient to explain performance differences. To provide deeper insight into program behavior, `benchkit` integrates the Linux `perf` suite, the standard tool for collecting run-time profiling and performance statistics.

`perf stat` provides access to hardware Performance Monitoring Counters (PMCs) and reports metrics such as instruction counts, CPU cycles, cache misses, branch mispredictions, context switches, and migrations. For example, invoking `perf stat -e cache-misses,cycles,instructions` around a benchmark run yields a detailed breakdown of the CPU activity during that run.

In `benchkit`, `perf` is integrated using a `CommandWrapper`. This wrapper automatically prepends the appropriate `perf stat` command to the benchmark invocation, ensuring that performance counters are collected transparently for each run, without having to modify the benchmark code. The wrapper is configured such that

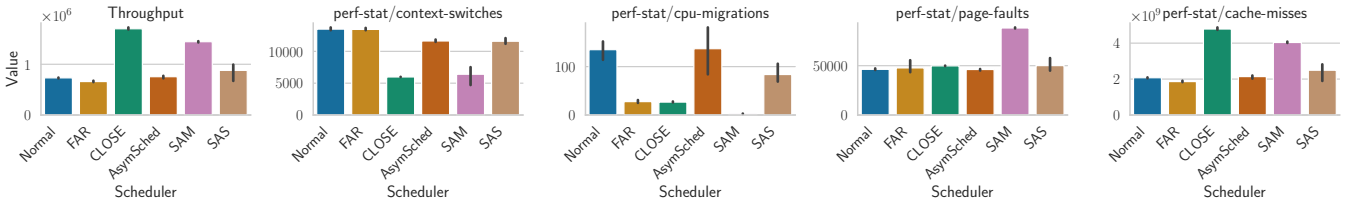


Figure 5: Performance analysis using `perf-stat` for the *LevelDB: readrandom* benchmark with 24 threads under various scheduling policies (3 runs of 30 seconds each). Throughput correlates strongly with the reduction in context switches and CPU migrations, particularly for the CLOSE policy, which pins threads and minimizes overhead. High cache misses for CLOSE and SAM indicate increased work completed during the fixed-duration benchmark. SAM incurs the highest page faults but maintains competitive throughput, suggesting effective sharing-aware mapping.

```python
from benchkit import CampaignCartesianProduct
from benchkit.benches.leveldb import LevelDBBench
from benchkit.commandwrappers.perf import PerfStatWrap
from benchkit.platforms import get_current_platform

platform = get_current_platform()
schedkit = SchedKitHooks(platform=platform)

schedulers = ["Normal", "FAR", "CLOSE",
              "AsymSched", "SAM", "SAS"]

perf_stat = PerfStatWrap(
    events=["context-switches", "cpu-migrations",
            "page-faults", "cache-misses"],
)

campaign = CampaignCartesianProduct(
    benchmark=LevelDBBench(),
    variables={
        "nb_threads": [24],
        "bench_name": ["readrandom"],
        "scheduler": schedulers,
    },
    nb_runs=3, duration_s=30,
    command_wrappers=[perf_stat],
    pre_run_hooks=[schedkit.start_sched_hook],
    post_run_hooks=[
        schedkit.end_sched_hook,
        perf_stat.post_run_hook_update_results,
    ],
)
campaign.run()
```

Listing 8: `benchkit` campaign for LevelDB with different scheduling policies and using `perf stat`.

the output file produced by `perf stat` (via its `-o` option) is saved inside the dedicated per-run directory created by `benchkit`, following the directory hierarchy introduced in Section 2.4. A corresponding `PostRunHook` then parses this file and inserts the extracted PMC metrics directly into the result record.

As a consequence, hardware-counter measurements become first-class experimental output metrics: they are collected consistently across all configurations, stored in the per-run directory hierarchy, and included in the CSV and JSON artifacts produced by the campaign, together with the metrics produced by the benchmark itself in its `collect()` method.

This wrapper+hook modeling enables `benchkit` to attach profiling to any benchmark—databases, memory allocators, synchronization stress tests, or system-level studies—with minimal user effort. Although two components are involved (the wrapper itself and its associated post-run hook), `benchkit` allows the hook to be referenced directly as a method of the wrapper instance (e.g., `perf_stat.post_run_hook_update_results`), which Python treats as a closure (or bound method)—implicitly currying the `self` pointer.

Listing 8 illustrates how `perf stat` can be combined with scheduling hooks to evaluate placement policies while collecting hardware performance counters in a single experiment. The example shows a `benchkit` campaign sweeping over several scheduling policies for the *LevelDB: readrandom* workload while collecting cache- and migration-related statistics. The aggregated results are reported in Figure 5.

4.4 Visualizing Performance with Flame Graphs in `benchkit`

While `perf stat` exposes high-level hardware counters, understanding *why* two configurations behave differently requires inspecting their execution profiles. To support such investigations, `benchkit` integrates `perf record` and provides native support for both standard and differential flame graphs.

Flame graphs aggregate stack samples into a folded call graph whose flame "width" represents relative time spent in each function, making hot paths and bottlenecks visually apparent [23, 24, 26]. Differential flame graphs extend this visualization by contrasting two runs and highlighting functions whose relative contribution increases or decreases between them [25].

Flame-graph generation maps naturally to `benchkit`'s experiment model: a *flame graph* corresponds to the execution profile of a *single record* in the experiment's parameter space, whereas a *differential flame graph* compares *two* records, typically differing in one parameter (e.g., lock type).

When the `PerfRecordWrap` wrapper is enabled in the campaign, each run is executed under `perf record`, and the `perf.data` output file is stored automatically inside the run's directory (as described in Section 2.4). After the run, a post-run hook processes this file to produce the flame graph for that configuration. Because every run is handled this way, users may later select any two records from the campaign and ask `benchkit` to generate a differential flame graph between them—a capability particularly useful during performance-debugging drilldowns.

The integration mirrors the modeling used for `perf stat`: the `PerfRecordWrap` class implements both the command-wrapper

interface—prepending `perf record` to the benchmark invocation and storing the profile in the result directory—and post-run hook methods that plug into the campaign, e.g., a flamegraph hook that transforms the recorded profile into a folded stack trace and then into a flame graph, or a report hook that invokes `perf report` for symbol-level analysis (see Listing 9). The class also provides a helper method for generating differential flame graphs, invoked *after* the campaign has completed. This helper takes as input two records (i.e., two variable–value mappings from the parameter space), locates the corresponding `perf.data` files in the directory hierarchy, and produces the differential flame graph as an additional artifact. In practice, this can be combined with campaign utilities (such as programmatic iteration over variables) to quickly compare two selected configurations.

Listing 9 shows how flame-graph profiling is integrated into a `benchkit` campaign comparing two lock implementations (CAS and MCS) for the *LevelDB: readrandom* workload under a fixed placement policy. Because flame graphs describe the behavior of a *single* run, external sources of variability—in particular thread placement—must be kept stable. Instead of manually pinning threads (which provides limited control for multithreaded applications), we simply *fix* the scheduling policy using the *CLOSE* placement strategy introduced in Section 4.2. This isolates the lock mechanism as the only varying parameter, making the comparison meaningful. The resulting flame graphs and their differential comparison are shown in Figure 6, illustrating how `benchkit` helps explain performance differences by visualizing hot paths and contention points.

```python
from benchkit import CampaignCartesianProduct
from benchkit.benches.leveldb import LevelDBBench
from benchkit.commandwrappers.perf import PerfRecordWrap
from benchkit.platforms import get_current_platform
from benchkit.sharedlibs.tiltlib import TiltLib

platform = get_current_platform()
schedkit = SchedKitHooks(platform=platform)

perf_record = PerfRecordWrap()

campaign = CampaignCartesianProduct(
    benchmark=LevelDBBench(),
    variables={
        "nb_threads": [32],
        "bench_name": ["readrandom"],
        "scheduler": ["CLOSE"],
        "lock": ["caslock", "mcslock"],
    },
    nb_runs=1, duration_s=10,
    shared_libs=[TiltLib()],
    command_wrappers=[perf_record],
    pre_run_hooks=[schedkit.start_sched_hook],
    post_run_hooks=[
        schedkit.end_sched_hook,
        perf_record.post_run_hook_report,
        perf_record.post_run_hook_flamegraph,
    ],
)
campaign.run()

# Generate a differential flame graph between CAS and MCS:
perf_record.generate_diff_flamegraph(
    campaign=campaign,
    record_a={"nb_threads": 32,
              "bench_name": "readrandom",
              "scheduler": "CLOSE",
              "lock": "caslock"},
    record_b={"nb_threads": 32,
              "bench_name": "readrandom",
              "scheduler": "CLOSE",
              "lock": "mcslock"},
    output="diff-cas-vs-mcs.svg",
)
```

Listing 9: `benchkit` campaign using `perf record` to generate flame graphs.

5 Overhead of `benchkit`

A performance evaluation framework is only useful if its own overhead is negligible. To assess this, we compare the end-to-end cost of running a benchmark through `benchkit` against a manually written shell script that reproduces the same sequence of build and run steps. We perform this comparison both on the native host and inside a Docker container, using `benchkit`'s native support for containerized execution (Section 2.4).

The experiment runs the *LevelDB: readrandom* benchmark under three thread counts (2, 4, 8), ten repetitions each, for 10 s per run. All measurements were collected on an Intel Core i7-13800H laptop (Ubuntu 22.04.5, Linux 6.8.0), using both native host and Docker execution.

We compare four configurations: `benchkit` running on the host, `benchkit` running in Docker, a hand-written shell script on the host, and the same script in Docker. The shell workflows duplicate exactly what the `benchkit` campaigns perform: building `db_bench`, running a single `fillseq` initialization, and evaluating `readrandom` with the same parameters, ensuring strict functional equivalence.

Figure 7 reports the throughput across all runs. Measurements cluster tightly, and the differences between the four setups are within natural run-to-run variability. On the host, the average throughput of the `benchkit` campaign differs from the equivalent hand-written shell workflow by only 2.22 % at 2 threads, 1.39 % at 4 threads, and 0.92 % at 8 threads. In Docker, the gaps are even smaller: 0.13 % at 2 threads, 0.66 % at 4 threads, and 0.68 % at 8 threads.

Across all scenarios, we observe no slowdown introduced by `benchkit`, no measurable overhead from the `Platform` abstraction, and no deviation in variance patterns compared to the shell workflows. The dominant fluctuations arise from LevelDB itself, not from the orchestration mechanism.

These results confirm that `benchkit` introduces *no measurable overhead* compared to equivalent hand-written scripts—neither on native hosts nor inside containers. This makes `benchkit` suitable for high-fidelity performance evaluation, including microbenchmarking and sensitivity analysis, where additional framework overhead would otherwise confound measurements.

6 Related Work

Reproducibility and artifacts. Reproducing systems performance results remains difficult even with artifacts, as studies by Collberg et al. [8] and Jiménez et al. [33] show that undocumented environment assumptions, ad-hoc orchestration, and non-portable scripts routinely prevent reliable reruns.

Benchmarking automation tools. Faban [18] provides automated benchmark orchestration with a Java-based harness and workload-driver framework supporting distributed load generation,

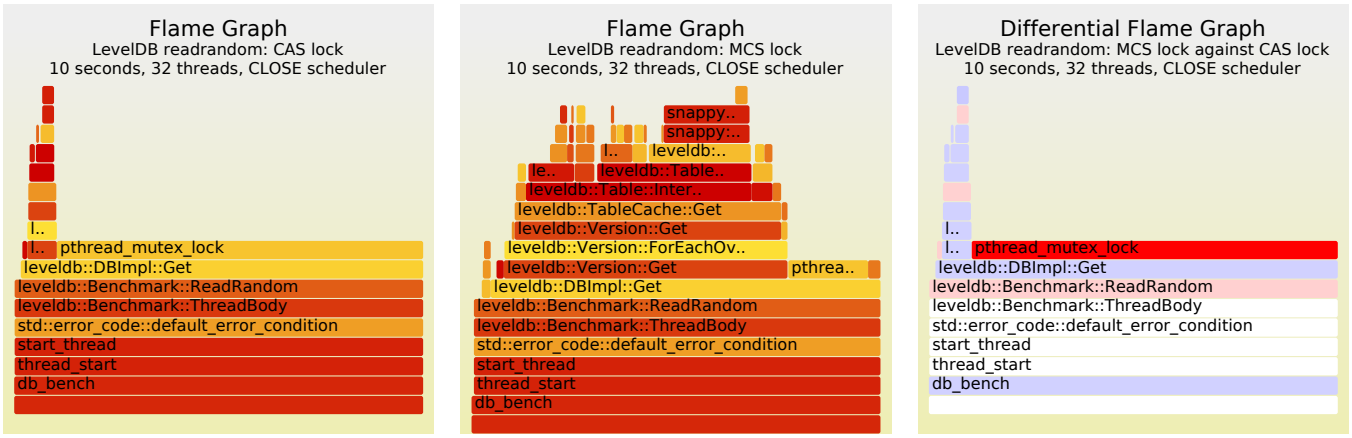


Figure 6: Flame graphs for CAS lock (*left*), MCS lock (*center*), and the differential flame graph between the two (*right*). In the regular flame graph, the color does not denote a specific metric but is used to visually distinguish different stack frames. In the differential flame graph, conversely, the color represents the difference in execution time between two configurations: red indicates an increase, while blue signifies a decrease. In this case, we observe that more time is spent in the `pthread_mutex_lock` call for the CAS lock compared to the MCS lock.

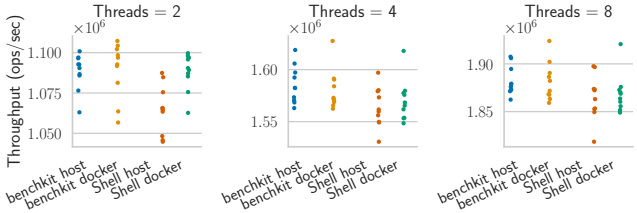


Figure 7: Throughput comparison between `benchkit` and manually crafted shell scripts, on host and inside Docker. Each point corresponds to one run. Measurements cluster tightly across all configurations, confirming that `benchkit` introduces no measurable overhead. Results shown for an Intel Core i7-13800H laptop.

but its annotation-driven model targets client–server workload simulation rather than system-level experimentation. Database-focused frameworks such as BenchBase [14] provide unified benchmarking across DBMSs, but are tightly coupled to database workloads and offer limited extensibility beyond that domain. Hyperfine [51] offers a lightweight CLI interface for microbenchmark timing but lacks support for NUMA-aware execution, affinity control, or profiling pipelines. The Collective Knowledge (CK) framework [19] emphasizes reproducible workflows and portable benchmarking components, primarily in the context of ML and compiler research. In the HPC domain, ReFrame [34] provides a regression-testing framework for validating system health across clusters, while JUBE [41] supports parameter-study workflows with Cartesian-product exploration; both are designed for HPC job schedulers and do not target bare-metal system-level profiling. Other specialized systems, such as PyTorch Benchmark [9], Optuna [2], or domain-specific benchmarks (e.g., schedbench [28] for scheduler stress tests), target narrow ecosystems and do not offer a general-purpose mechanism for combining performance experiments with placement, instrumentation, and profiling. In contrast, `benchkit` provides a lightweight, extensible foundation for heterogeneous systems studies: it supports arbitrary benchmarks, custom parameters, hierarchical result storage, and composable orchestration through hooks, command wrappers, and shared-library injection.

System-level performance profilers and analysis tools. Low-level profilers such as `perf`, Intel VTune, and AMD uProf provide rich access to hardware counters, flame graphs, and microarchitectural statistics, but do not prescribe how profiling should integrate into multi-configuration experiments, nor how to guarantee consistent environmental conditions across runs. Similarly, tools in the GPU ecosystem—such as NVIDIA Nsight Compute and Nsight Systems—offer detailed traces of GPU kernels but require manual setup and do not automate placement, repetition, or post-run aggregation. sysperf [32] and related utilities focus on system-call tracing or profiling small program variants, but do not treat experiment orchestration or parameter–space exploration as first-class concepts. `benchkit` does not replace these profilers; rather, it provides a structured way to embed them into repeatable experiments, ensuring that profiling steps (e.g., `perf stat`, `perf record`) are executed deterministically and that their outputs become part of the experiment's artifact set.

Cloud infrastructure benchmarking. Cloud-native platforms have motivated the development of benchmarking tools that operate within container orchestrators. Kubestone [60] defines benchmarks as Kubernetes Custom Resources and leverages operators for scheduling and execution. Cloud-Native-Bench [59] provides a Kubernetes-operator-based framework with automated data gathering and analysis for cloud performance testing. Henning and Hasselbring [29] propose a configurable method for benchmarking scalability of cloud-native applications deployed on Kubernetes clusters. Papadopoulos et al. [50] define methodological principles for reproducible performance evaluation in cloud environments.

These tools are tightly coupled to container orchestration and target cloud-service workloads; they do not support bare-metal concerns such as NUMA-aware placement, shared-library injection, or fine-grained profiling pipelines that `benchkit` targets. Conversely, `benchkit` does not yet offer first-class Kubernetes orchestration (see Section 7).

Positioning of `benchkit`. Where existing tools typically focus on a single concern—database benchmarks, microbenchmark repetition, workflow packaging, or low-level profiling—`benchkit` provides a unifying abstraction for *composable* systems experiments. Its hook-based modeling supports thread-placement daemons, shared-library lock frameworks, performance counters, profiling traces, flame-graph generation, and containerized execution inside a single, coherent experiment definition. This composability, combined with its declarative parameter–space exploration and reproducibility-oriented result hierarchy, distinguishes `benchkit` from prior work and enables systematic, multi-dimensional performance studies hardly achievable with existing benchmarking automation tools. Earlier versions of `benchkit` have already been used to evaluate synchronization primitives on large NUMA servers [10, 45, 46].

7 Limitations

Wrapper engineering overhead. Using `benchkit` requires implementing Python classes for new benchmarks and command wrappers, which introduces an upfront development cost compared to ad-hoc shell scripts. However, only the `run()` method is mandatory for a benchmark class, and lightweight helpers exist for wrapping single-command workloads with minimal boilerplate. Crucially, once a benchmark or wrapper is implemented, it is immediately reusable across arbitrarily different experiments without modification: a `TasksetWrap` written once can be composed with any benchmark, any parameter space, and any set of hooks or profiling tools. This *write-once, compose-anywhere* property is where `benchkit` provides its greatest return over shell scripts, which typically require per-experiment adaptation and offer limited modularity when the number of variables and tools grows.

Portability. The `benchkit` orchestration layer is pure Python and can run on any platform, including Windows or macOS, provided the target system (local, remote via SSH, or mobile via ADB) supports shell command execution. However, many of the built-in wrappers and hooks (`perf`, `taskset`, `numactl`, `LD_PRELOAD`) are Linux-specific capabilities. More fundamentally, hardware performance counters and kernel interfaces are architecture- and version-specific: even for the same concept (e.g., last-level cache misses), the `perf` event name differs across microarchitectures—e.g., `l3d_cache_refill` on ARM Cortex-A76, `l3_misses` on AMD Zen 5, or `mem_bound_stalls.load_llc_hit` on Intel. A campaign that specifies raw event names may therefore fail or produce non-comparable results on a different microarchitecture. Nothing in the framework's design prevents the introduction of abstraction layers that map portable, architecture-neutral event names to platform-specific counters—similar to how `TasksetWrap` already abstracts CPU-list syntax (see Listing 4). Building and maintaining such a mapping across processor generations remains an open engineering challenge.

Scope. `benchkit` supports multi-node setups where, e.g., a database server and its load generator run on separate machines (each modeled as a distinct platform). However, it does not yet offer first-class orchestration for large-scale distributed deployments such as Kubernetes clusters. Additionally, campaigns assume a single fetch per benchmark; regression workflows that test multiple versions of the same software (e.g., across repository commits) would benefit from parameterizing the fetch step itself. Finally, campaigns currently execute runs sequentially—a deliberate design choice rooted in `benchkit`'s original focus on multithreaded workloads where interference-free, full-machine runs are essential—but alternative execution engines, such as parallel dispatch of independent single-threaded jobs, are under active development.

8 Conclusion

This paper introduced `benchkit`, a lightweight yet expressive framework that brings software-engineering principles—abstraction, composition, and explicit modeling of system interactions—into performance evaluation. By treating benchmarks, parameter spaces, wrappers, hooks, and shared libraries as first-class objects, `benchkit` replaces ad-hoc scripting with declarative, repeatable, composable, and extensible experiment definitions.

Although `benchkit` provides built-in integrations (e.g., `perf stat`, `perf record`, thread placement, containerized execution), its true strength lies in extensibility: users can add custom benchmarks, platforms, wrappers, and shared libraries to express complex system-level experiments without modifying benchmark code. Our own tools, `tilt` (lock interposition) and `schedkit` (plugin-based thread placement), are available as open source to support replication and further study.

Future directions include integrating speedup stacks [53] for structured scalability analysis, adding eBPF-based instrumentation, and enriching visualizations—for example, generating per-run Gantt charts that show *when and on which core* each thread of the benchmark executed [36]. Overall, `benchkit` aims to provide a principled foundation for systematic and reproducible systems performance evaluation.

Acknowledgment

We thank the anonymous reviewers for their helpful comments, Peter Buhr for providing access to large NUMA machines, Seppe Wyns and Thomas Vandermotten for the initial co-development of `schedkit`, and Huawei Dresden Research Center for supporting the initial public release of `benchkit` as an open-source project under the MIT license. This work has been supported in part by the Royal Higher Institute for Defence (RHID) under the Defence-related Research Action (DEFRA), contract no. 24DEFRA001, and Research Foundation Flanders (FWO) under grant no. G096225N.

References

[1] ACM. 2020. Artifact Review and Badging, Version 1.1. https://www.acm.org/publications/policies/artifact-review-and-badging-current Accessed: 2025-01-15.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2623–2631. doi:10.1145/3292500.3330701

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 158–165. doi:10.1145/125826.125925

[4] Anton Blanchard. 2020. will-it-scale. https://github.com/antonblanchard/will-it-scale. Accessed: 2026-02-20.

[5] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3185768.3185771

[6] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. *SIGPLAN Not.* 50, 8 (2015), 215–226. doi:10.1145/2858788.2688503

[7] Andy Cockburn, Pierre Dragicevic, Lonni Besançon, and Carl Gutwin. 2020. Threats of a Replication Crisis in Empirical Computer Science. *Commun. ACM* 63, 8 (2020), 70–79. doi:10.1145/3360311

[8] Christian Collberg and Todd A. Proebsting. 2016. Repeatability in computer systems research. *Commun. ACM* 59, 3 (Feb 2016), 62–69. doi:10.1145/2812803

[9] Will Constable, Xu Zhao, Victor Bittorf, Eric Christoffersen, Taylor Robie, Eric Han, Peng Wu, Nick Korovaiko, Jason Ansel, Orion Reblitz-Richardson, and Soumith Chintala. 2020. TorchBench: A collection of open source benchmarks for PyTorch performance and usability evaluation. https://github.com/pytorch/benchmark. Accessed: 2025-11-19.

[10] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. 2021. CLoF: A Compositional Lock Framework for Multi-level NUMA Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 851–865. doi:10.1145/3477132.3483557

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan 2008), 107–113. doi:10.1145/1327452.1327492

[12] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 12, 15 pages. doi:10.1145/3302424.3303984

[13] Dave Dice and Alex Kogan. 2021. Hemlock: Compact and Scalable Mutual Exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 173–183. doi:10.1145/3409964.3461805

[14] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[15] Dormando. 2003. Memcached: High-Performance Distributed Memory Caching. https://memcached.org/. Accessed: 2025-11-19.

[16] Jason Evans. 2005. jemalloc: A General-Purpose Memory Allocator. https://github.com/jemalloc/jemalloc. Accessed: 2025-11-19.

[17] Facebook, Inc. 2013. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb. Accessed: 2025-11-19.

[18] Vincenzo Ferme and Cesare Pautasso. 2016. Integrating Faban with Docker for Performance Benchmarking. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 129–130. doi:10.1145/2851553.2858676 Faban source: https://github.com/akara/faban.

[19] Grigori Fursin. 2021. Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common interfaces. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379, 2197 (2021), 20200211. doi:10.1098/rsta.2020.0211

[20] Mikio Fushima. 2009. Kyoto Cabinet: A Modern Implementation of the DBM Database. https://dbmx.net/kyotocabinet/. Accessed: 2025-11-19.

[21] Google, Inc. 2006. TCMalloc: Thread-Caching Malloc. https://github.com/google/tcmalloc. Accessed: 2025-11-19.

[22] Google, Inc. 2011. LevelDB: A Fast and Lightweight Key/Value Database Library. https://github.com/google/leveldb. Accessed: 2024-12-19.

[23] Brendan Gregg. 2013. Flame Graphs. http://www.brendangregg.com/flamegraphs.html. Accessed: 2024-12-18.

[24] Brendan Gregg. 2013. *Systems Performance: Enterprise and the Cloud*. Prentice Hall, Upper Saddle River, NJ, USA.

[25] Brendan Gregg. 2014. Differential Flame Graphs. https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html. Accessed: 2024-12-18.

[26] Brendan Gregg. 2017. Visualizing Performance with Flame Graphs. Invited Talk, USENIX ATC '17. https://www.usenix.org/conference/atc17/program/presentation/gregg-flame.

[27] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case Is Not Closed Yet. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 649–662. https://www.usenix.org/conference/atc16/technical-sessions/presentation/guiroux

[28] HashiCorp. 2016. schedbench. https://github.com/hashicorp/schedbench. Accessed: 2026-02-20.

[29] Sören Henning and Wilhelm Hasselbring. 2022. A Configurable Method for Benchmarking Scalability of Cloud-Native Applications. *Empirical Software Engineering* 27, 6 (2022), 143. doi:10.1007/s10664-022-10162-1

[30] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[31] Hugo Guiroux. 2016. LiTL: Library for Transparent Lock interposition. https://github.com/multicore-locks/litl. Accessed: 2024-12-18.

[32] iandk (GitHub user). 2024. sysperf: A lightweight benchmark script to test your Disk, Network and CPU performance. https://github.com/iandk/sysperf. Accessed: 2025-11-20.

[33] Ivo Jimenez, Michael Sevilla, Noah Watkins, and Carlos Maltzahn. 2016. *Popper: Making Reproducible Systems Performance Evaluation Practical*. Technical Report UCSC-SOE-16-10. University of California, Santa Cruz.

[34] Vasileios Karakasis, Theofilos Manitaras, Victor Holanda Rusu, Rafael Sarmiento-Pérez, Christopher Bignamini, Matthias Kraushaar, Andreas Jocksch, Samuel Omlin, Guilherme Peretti-Pezzi, João P. S. C. Augusto, Brian Friesen, Yun He, Lisa Gerhardt, Brandon Cook, Zhi-Qiang You, Samuel Khuvis, and Karen Tomko. 2020. Enabling Continuous Testing of HPC Systems Using ReFrame. In *Tools and Techniques for High Performance Computing*, Guido Juckeland and Sunita Chandrasekaran (Eds.). Springer International Publishing, Cham, 49–68. doi:10.1007/978-3-030-44728-1_3

[35] Alexey Kopytov and contributors. 2004. sysbench: Scriptable Database and System Benchmark Tool. https://github.com/akopytov/sysbench. Accessed: 2025-11-19.

[36] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, and Gilles Muller. 2023. *Graphing Tools for Scheduler Tracing*. Technical Report RR-9498. Inria Paris. https://inria.hal.science/hal-04001993

[37] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 277–289. https://www.usenix.org/conference/atc15/technical-session/presentation/lepers

[38] Linux Kernel Developers. 2008. Linux Ticketlock Implementation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=314cdbefd1fd0a7acf3780e9628465b77ea6a836. Accessed: 2024-12-19.

[39] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. doi:10.1145/2901318.2901326

[40] Redis Ltd. 2009. Redis: An In-Memory Key-Value Store. https://redis.io/. Accessed: 2025-11-19.

[41] Sebastian Lührs, Daniel Rohe, Alexander Schnurpfeil, Kay Thust, and Wolfgang Frings. 2016. Flexible and Generic Workflow Management. In *Parallel Computing: On the Road to Exascale* (2015-09-01) *(Advances in parallel computing, Vol. 27)*. International Conference on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep 2015, IOS Press, Amsterdam, 431–438. doi:10.3233/978-1-61499-621-7-431

[42] Paul E. McKenney, Davidlohr Bueso, and contributors. 2014. locktorture: Kernel Locking-Primitive Torture Test Module. https://www.kernel.org/doc/html/latest/locking/locktorture.html. Accessed: 2024-12-19.

[43] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65. doi:10.1145/103727.103729

[44] Nature Publishing Group. 2021. But is the code (re)usable? *Nature Computational Science* 1, 7 (2021), 449–449. doi:10.1038/s43588-021-00109-9

[45] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 530–545. doi:10.1145/3445814.3446748

[46] Jonas Oberhauser, Lilith Oberhauser, Antonio Paolillo, Diogo Behrens, Ming Fu, and Viktor Vafeiadis. 2021. Verifying and Optimizing the HMCS Lock for Arm Servers. In *Networked Systems: 9th International Conference, NETYS 2021, Virtual Event, May 19–21, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 240–260. doi:10.1007/978-3-030-91014-3_17

[47] open-s4c. 2024. libvsync: A verified library of synchronization primitives and concurrent data structures. https://github.com/open-s4c/libvsync. MIT License.

Accessed: 2026-02-19.

[48] Oracle Corporation. 1995. MySQL: The World's Most Popular Open Source Database. https://www.mysql.com/. Accessed: 2025-11-19.

[49] Antonio Paolillo. 2026. pythainer: composable and reusable Docker builders and runners for reproducible research. *Journal of Open Source Software* 11, 117 (2026), 9059. doi:10.21105/joss.09059

[50] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, José Nelson Amaral, Petr Tůma, and Alexandru Iosup. 2021. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1528–1543. doi:10.1109/TSE.2019.2927908

[51] David Peter. 2023. hyperfine. https://github.com/sharkdp/hyperfine.

[52] Phoronix. 2008. Phoronix Test Suite: Comprehensive Open-Source Benchmarking Platform. https://www.phoronix-test-suite.com/. Accessed: 2024-12-19.

[53] Jennifer B. Sartor, Kristof Du Bois, Stijn Eyerman, and Lieven Eeckhout. 2017. Analyzing the scalability of managed language applications with speedup stacks. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Santa Rosa, CA, USA, 23–32. doi:10.1109/ISPASS.2017.7975267

[54] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2015. Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems. In *USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 529–540.

[55] David Tam, Reza Azimi, and Michael Stumm. 2007. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *SIGOPS Oper. Syst. Rev.* 41, 3 (2007), 47–58. doi:10.1145/1272998.1273004

[56] Nick Tehrany. 2020. membench: Benchmarking Memory Bandwidth/Latency, Page-Fault, and mmap Latency. https://github.com/nicktehrany/membench. Accessed: 2025-11-19.

[57] The Linux Foundation. 2025. Cyclictest — realtime documentation howto tools. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start. Accessed 2025-08-07.

[58] The PostgreSQL Global Development Group. 1996. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/. Accessed: 2025-11-19.

[59] Michiel Van Kenhove, Merlijn Sebrechts, Filip De Turck, and Bruno Volckaert. 2023. Cloud-Native-Bench: an Extensible Benchmarking Framework to Streamline Cloud Performance Tests. In *2023 IEEE 12th International Conference on Cloud Networking (CloudNet)*. IEEE, Hoboken, NJ, USA, 86–93. doi:10.1109/CloudNet59005.2023.10490079

[60] xridge.io. 2019. Kubestone: Performance Benchmarks for Kubernetes. https://github.com/kubestone/kubestone. Apache-2.0 License. Accessed: 2026-02-19.