Reproducible Performance Evaluation of OpenMP and SYCL Workloads under Noise Injection

Reproducible Performance Evaluation of OpenMP and SYCL Workloads under Noise Injection

Christoffer Persson*
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
o.christoffer.persson@gmail.com

Mathias Pretot*
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
mathias.pretot@gmail.com

Minyu Cui
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
minyu@chalmers.se

Miquel Pericàs
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
miquelp@chalmers.se

Abstract

Performance instability caused by unpredictable system noise remains a persistent challenge in high-performance and parallel computing. This work presents a reproducible methodology to characterize this variability through noise injection, tested using workloads implemented in OpenMP and SYCL to compare their performance resilience under noisy conditions. We design a noise injector that captures real system traces and replays the deltas as controlled noises. Using this approach, we evaluate multiple mitigation efforts, that is, thread pinning, housekeeping core isolation, and simultaneous multithreading (SMT) toggling, under both default and noise-injected executions. Experiments with two benchmarks (N-body, Babelstream) and one mini-application (MiniFE) across two processor platforms show that while OpenMP consistently achieves higher raw performance, SYCL tends to exhibit greater resilience in noisy environments. Mitigation effectiveness varies with workload characteristics, system configuration, and noise intensity, with housekeeping core isolation offering the clearest benefits, particularly in high-noise scenarios.

CCS Concepts

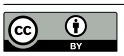
• **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → **Parallel programming languages**.

Keywords

Parallel Computing, Performance Variability, Mitigation Strategies, Noise Injection, OpenMP, SYCL

ACM Reference Format:
Christoffer Persson, Mathias Pretot, Minyu Cui, and Miquel Pericàs. 2025. Reproducible Performance Evaluation of OpenMP and SYCL Workloads under Noise Injection. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767538

*These authors contributed equally.



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767538

1 Introduction

In high-performance computing (HPC), performance instability in parallel applications is a well-recognized concern in most computing systems. Fluctuations in execution time complicate the accurate prediction of the time frame for parallel programs, both across platforms and between repeated runs on the same platform, particularly for long-running workloads. Neglecting this variability can lead to systematic underestimation of execution time, especially when systems frequently approach worst-case behavior. For example, while a 10% delay between runs is negligible for a task of one second, it can result in substantial delays for workloads with execution time of one week. Moreover, reducing execution time variability and achieving stable average throughput are crucial for efficient software development. When discrepancies in execution times are minimized, code optimizations can be assessed more easily, as the influence of uncontrollable external factors is reduced [4]. Performance stability is particularly important for applications with interactivity [17] or strict timing requirements[14], where predictable execution behavior is essential.

Operating system (OS) noise [19, 22] is widely acknowledged as a primary source of performance variability. This noise originates from OS daemons and background processes competing for shared hardware resources. Mitigation strategies aim to reduce execution time fluctuations between workload runs and generally include both algorithmic changes and system-level configuration adjustments. A common evaluation method is to iteratively execute workloads over an extended period under mitigation effort and then compare the results against a baseline. While accurate, this approach is sensitive to the total execution time and the stochastic occurrence of rare system events. Furthermore, the mitigation itself may inadvertently suppress the original behaviors it is intended to address, complicating direct comparisons. This raises two main challenges: (1) ensuring comparable levels of system noise across runs and (2) requiring extended execution times to capture the relevant variability phenomena.

OpenMP and SYCL are two widely adopted programming models for parallel computing. OpenMP [20] extends C, C++, and Fortran with compiler directives, runtime routines, and environment variables, enabling shared-memory parallelism and offloading with minimal code changes. SYCL [13] is a C++-based, single-source model standardized by the Khronos Group, offering cross-platform portability across CPUs, GPUs, and FPGAs. While the performance characteristics of OpenMP workloads has been extensively studied, research on performance variability in SYCL applications remains limited. To the best of our knowledge, no prior systematic investigation has been examined how system noise affects its performance stability, nor have methods to reproducibly mitigate variability been thoroughly evaluated. This leaves a gap in understanding how SYCL applications respond to system-level noise. This work conducts a comparative study of performance variability in workloads implemented in OpenMP and SYCL.

Existing studies on performance variability focus mainly on characterizing the variability induced by natural system-level noise [3, 6, 7, 22]. This reliance on inherent noise limits reproducibility and experimental control, making it challenging to evaluate mitigation strategies under consistent conditions. Unlike natural system noise, which is inherently unpredictable, controlled noise introduced by noise injection [2, 9] can be tailored to evaluate specific scenarios. These noise injectors typically either generate synthetic noise like HPAS (High Performance Anomaly Suite) in [2] or rely on specific simulation-based approaches [9]. However, they fail to capture the complexity or variability of real-world system noise. To address this limitation, this work presents the design and deployment of a CPU-based noise injector that reproduces noise patterns captured from real systems in operation. By injecting these observed system noise during program execution in a controlled and repeatable manner, our approach enables a rigorous evaluation of noise-mitigation strategies under worst-case scenarios, without relying on stochastic occurrences of rare system events.

This work reproducibly evaluates the performance variability in parallel programs under controlled system noise using a custom noise injector. The key contributions are as follows:

(1) **Design and deployment of a noise injector:** This work designs a more realistic noise-injection approach that replays traces of real-world system activities rather than generating purely synthetic noises. Our method better captures the unpredictable and dynamic nature of the system noise. This design delivers a deeper understanding of application behavior under realistic operating conditions and overcomes key limitation of synthetic noises such as HPAS [2].
(2) **Evaluation of performance variability mitigation strategies:** This work conducts a systematic evaluation of multiple mitigation efforts, such as thread-pinning, reserving housekeeping cores and SMT toggling, using two benchmarks (N-body, Babelstream) and one mini-application (MiniFE) across several multi-core production platforms. We assess the extent to which these strategies reduce performance variability and validate the noise injector's ability to reproduce worst-case conditions across different implementations.
(3) **Comparative insights into OpenMP and SYCL implementations:** This work investigates the performance variability of workloads implemented in both OpenMP and SYCL. Although OpenMP has been extensively studied in this context, SYCL's behavior regarding performance variability remains largely unexplored. By running the workloads under controlled noise, we analyze the effectiveness of mitigation strategies in both environments. This comparison not only provides new insights into SYCL runtime behavior, but also demonstrates how controlled noise can be leveraged for systematic and reproducible evaluation of parallel programming models and runtime configurations.

Section 2 reviews related work, followed by a motivation example in Section 3. Section 4 describes our noise injector design. The experimental results are presented in Section 5, with discussion and recommendations in Section 6. We conclude this work and outline future directions in Section 7.

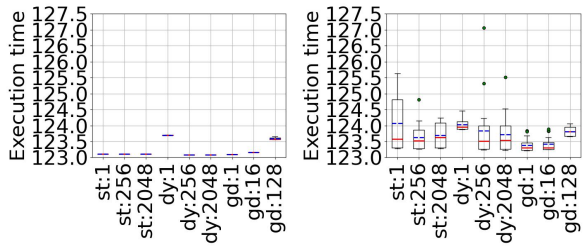
2 Related Work

System noise, also known as OS noise or jitter, is a major contributor to performance variability in parallel programs. It arises from OS activities such as task scheduling, interrupt handling, and I/O operations [8, 22]. Detecting and quantifying this variability is critical in modern HPC environments. Prior work has proposed tools for the detection of performance variation on large-scale heterogeneous systems [24], analyzing program execution by segmenting external library calls and asynchronous kernel operations. Lightweight OS kernels [23] minimize OS interference, enabling precise measurements of performance variability, particularly hardware-induced variation. This work highlights the need for accurate cross-platform characterization tools, as traditional OS-jitter benchmarks (e.g., FWQ) often fail to capture true hardware variability. Although inherent system noise is naturally occurring, unpredictable, and difficult to control, noise injection - the deliberate introduction of controlled disturbances into the system - enables systematic evaluation and reproducible testing of mitigation strategies [2, 9]. The authors [9] present a simulation-based toolchain that incorporates noise traces from large-scale architectures into LogGPS simulations, providing insights into application scaling under system noise. Similarly, HPAS [2] offers synthetic noise generators that emulate CPU occupation, memory bandwidth fluctuations, and other system disturbances, allowing repeated assessment of workload responses to varying resource contention.

While performance variability is broadly recognized, it is often treated as an afterthought. Mitigation strategies have been proposed to enhance the performance stability of parallel applications, such as fixing thread placement for OpenMP workloads [6, 7], reserving the core resource to absorb OS noise [7, 16, 21]. A variant of core reservation leverages the SMT feature by leaving the secondary hardware threads free for OS tasks [15]. Running workloads on both threads per core can increase variability compared to single-threaded execution [18]. A less common but highly effective mitigation is to dedicate specific cores exclusively to the OS, often rendering them invisible to users [1], which is a practice typically implemented by system vendors.

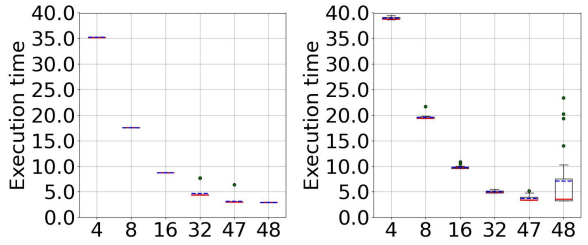
Performance variability in workloads implemented using OpenMP has been extensively investigated [3, 6, 7, 18, 21], whereas research on SYCL has largely concentrated on performance portability [10–12]. These studies primarily aim to ensure that SYCL applications can run efficiently and consistently across different backends without requiring substantial changes to the code. However, research specifically addressing performance variability in SYCL is largely absent.

3 Motivation Example



(a) A64FX: reserved (b) A64FX: w/o

Figure 1: Higher variability for *schedbench* without system reserved cores. The x-axis uses the *xy:number* format, where *xy* denotes the schedule methods, namely static (st), dynamic (dy) and guided (gd), and *number* specifies the chunk size.



(a) A64FX: reserved (b) A64FX: w/o

Figure 2: Higher variability for *dot* kernel from *Babelstream* without system reserved cores. The x-axis indicates the number of threads.

A widely adopted strategy for mitigating OS noise is core reservation, where a subset of CPU cores is dedicated to OS and service threads and made invisible to users. For example, Cray XE implements core specialization [21], and Fujitsu A64FX at the Barcelona Supercomputing Center reserves one or more cores for OS tasks. On the A64FX system with reserved cores (denoted as A64FX:reserved), two cores are reserved exclusively for OS tasks via a firmware-level configuration, which exposes these cores as separate NUMA nodes and hides them from user applications. This reservation strategy provides promising guarantees for performance stability. In contrast, an otherwise identical A64FX system without core reservation (denoted as A64FX:w/o) at the Minho Advanced Computing Center exhibits significantly higher performance variability. Figures 1 and 2 highlight this difference by comparing the execution times (ms) of *schedbench* and *Babelstream* across these two A64FX systems. The A64FX:reserved system demonstrates consistently lower execution time variability, whereas A64FX:w/o experiences substantially higher fluctuations, particularly when all 48 cores are fully utilized by user applications, as shown in Figure 2b. In this scenario, no spare cores remain to absorb OS interference, which in turn disrupts the execution of user workloads.

Such core reservation is typically employed sparingly because it reduces the total number of cores available to user applications and is often viewed as an inefficient use of computational resources. Consequently, most production systems have prioritized maximizing raw core availability over isolation for performance stability, leaving performance variability to be addressed as a secondary concern. Motivated by this limitation, this work focuses on systems without dedicated OS cores and investigates alternative mitigation strategies to reduce execution time variability.

4 Noise Injector

This section describes the design and operation of the proposed noise injector, which systematically reproduces system-level interference during workload execution. The injector operates in three key stages:

- **System trace collection**: The target workload is executed multiple times with system-level tracing enabled. This provides a dataset that captures both average system activity and instances of worst-case performance.
- **Noise configuration generation**: From the collected traces, the worst-performing execution is identified. To avoid double-counting background interference, the average system noise across all runs (indicating the inherent system noise), is subtracted from the traces. The resulting configuration file specifies the noise to inject.
- **Noise injection during workload execution**: The workload is re-executed while the noise injector replays the observed noise from the configuration file. This enables a controlled, repeatable emulation of worst-case system behavior, facilitating reproducible variability experiments.

The subsequent sections provide a detailed description of these stages. We note that the noise injection and workload tracing procedures described in this work require elevated privileges (e.g., root access). This is primarily due to the use of the `SCHED_FIFO` real-time scheduling policy, the disabling of system-level fail-safes for processor utilization, and the need to configure low-level tracing mechanisms.

4.1 System trace collection

The noise injection process begins by collecting system traces during 1000 workload executions with *OSnoise* [8] enabled, producing one trace file per execution. These traces establish baseline system behavior and identify the execution of a worst-case to emulate. Each trace records *OSnoise* events for every logical core. Figure 3 shows a simplified example of such trace. The *CPU* column specifies the core where the event occurred, and the *Event type* distinguishes between thread activity and interrupt handling. The *Source* identifies the process responsible for the event, such as `local_timer` or `kworker`. The *Start time* indicates the precise timestamp—relative

CPU	Event Type	Source	Start Time	Duration
005	irq_noise	local_timer:236	255.045740274	310 ns
010	softirq_noise	RCU:9	255.045742404	140 ns
025	softirq_noise	SCHED:7	255.045742554	690 ns
024	irq_noise	local_timer:236	256.100739459	170 ns
031	irq_noise	local_timer:236	256.100739459	180 ns
013	thread_noise	kworker/13:1	256.188747948	3760 ns
001	thread_noise	kworker/u129:5	256.188750718	5830 ns

Figure 3: Sample entries from the *OSnoise* trace, illustrating typical event records. The entries have been slightly simplified and aligned for easier readability.

to the beginning of the trace—at which the event began, while the *Duration* specifies how long the event lasted before completion. It is important to note that *OSnoise* labels all events as noise, as it cannot differentiate noise from background interference and workload execution.

The collected traces provide two insights: 1) the average system noise, obtained by averaging the frequency and duration of recurring tasks across all executions, and 2) the worst-case trace, identified as the execution with the longest execution time. While this work primarily focuses on the worst-case trace, the noise injector is capable of replaying any captured trace in principle.

4.2 Noise configuration generation

The second stage of the noise injector constructs a configuration file specifying the start times, durations, and scheduling policies of individual noise events to be injected. Using the traces gathered during the first stage, the average frequency and duration for each unique noise are calculated, forming a baseline of the average system noise which indicates the inherent noise. The worst-case execution trace is subsequently refined by subtracting these average per-task noise contributions. This adjustment prevents the double-counting of noise and avoids over-pessimistic injection scenarios since a baseline level of inherent noise may still be present during execution. The refinement process proceeds as follows: for each noise event in the worst-case trace, the instance whose duration is closest to the corresponding average is reduced by the average duration. As a result, the representative occurrence of an average task is removed, leaving only the residual "delta" noise to be injected. This procedure is repeated for as many occurrences as expected, based on the average frequency of the task within the worst-case execution window. Figure 4 illustrates this process where noise events of a single origin in the worst-case trace are systematically reduced according to their average frequency and amount.

To accurately reflect the interaction with the OS scheduling, each injected noise is assigned a scheduling policy based on its original classification in the *OSnoise* trace (Figure 3). Events labeled as `thread_noise` use the default Linux scheduling policy `SCHED_OTHER`, whereas the others, such as `irq_noise` and `softirq_noise`, are mapped to the real-time `SCHED_FIFO` policy. This mapping ensures that the injected noise interacts with the OS in a manner consistent with actual system behavior. Finally, a configuration file is generated, as illustrated in Figure 5. Each thread in the refined worst-case trace is mapped to a corresponding list of noise events, where each event is annotated with its start time, duration, and assigned scheduling policy. This configuration

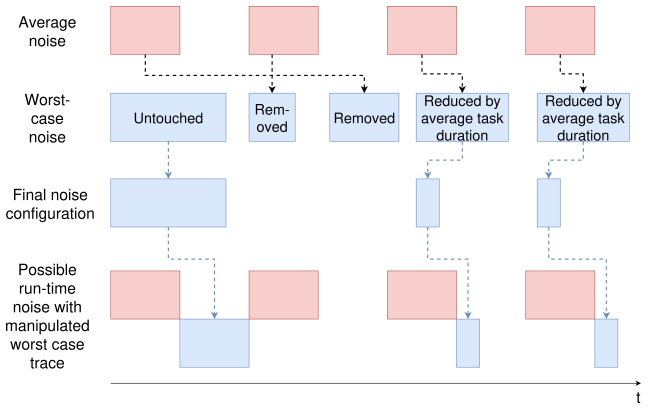


Figure 4: Schedule showing how the worst-case trace is altered by the average system noise and a possible runtime noise behaviour.

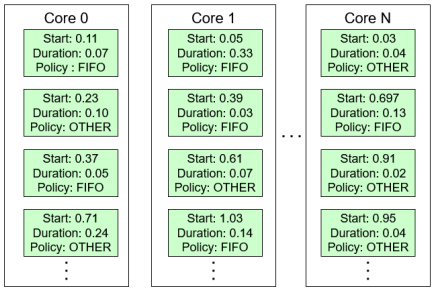


Figure 5: Overview of the generated configuration file structure

serves as the blueprint for precise and reproducible noise injection during workload execution.

4.3 Noise injection during workload execution

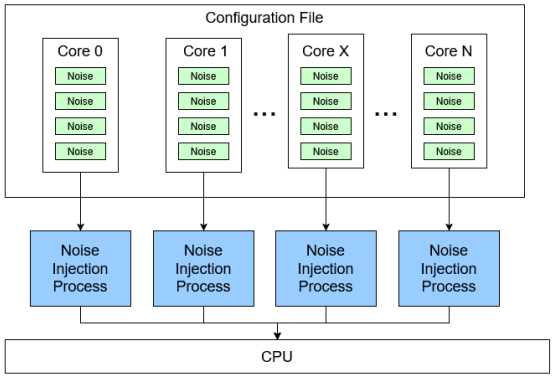


Figure 6: Overview of how the noise configuration is processed with the noise injector.

In the final stage, the noise injector configuration is used to introduce explicitly generated noise during workload execution, emulating worst-case system behavior observed in previous measurements. Although inherent noise remains an uncontrollable factor, leading to some variability cross executions, the average behavior experienced by the workload should approximate the intended worst-case scenario. The noise injection consists of a set of processes, each corresponding to a logical core specified in the noise configuration file which iterates over the noise events assigned to their respective cores, as shown in Figure 6. The total number of processes spawned is determined by the amount of logical cores specified in the noise configuration, typically matching the number of cores available in the system. To ensure flexibility when the workload does not execute on the exact cores associated with the emulated worst-case trace, the injected noise events can execute on any core as the spawned processes themselves do not have any processor affinities.

```
Input: config_file, processor_nr

noise_list <- ParseJSON(config_file, processor_nr)
SynchronizeWithPeers()

for each noise_event in noise_list do
    if CurrentPolicy() != noise_event.policy then
        SetPolicy(noise_event.policy)
    end if
    SleepUntil(noise_event.start_time)
    Inject(noise_event.duration)
end for
```

Listing 1: Pseudocode for noise injection

The pseudocode for the noise injection logic is shown in Listing 1. Each injection process follows a uniform routine. After initialization, all injector threads and the workload synchronize at a barrier to align their start times. Once the barrier is released, each injector sequentially processes its list of noise events. For each event, the injector first checks whether the current scheduling policy matches the one specified for the event, performing a policy switch if necessary. Then it sleeps until the event's scheduled start time, and upon waking, occupies the core for the specified duration. This procedure continues until all noise events have been injected or the workload signals completion, prompting early termination. Each noise injection process operates under either the default (`SCHED_OTHER`) or the real-time FIFO (`SCHED_FIFO`) scheduling policy. The `SCHED_FIFO` policy ensures strict priority over `SCHED_OTHER`, meaning that a regular workload cannot preempt, and is always preempted by noise processes running under `SCHED_FIFO`. This guarantees that noise events execute with a strictly adhered-to timing, closely mirroring the worst-case conditions. Furthermore, the default fail-safe, which limits CPU usage of FIFO scheduled processes, is explicitly disabled during noise injection to allow for 100% processor utilization.

5 Experimental Results

This section presents the results of evaluating noise injection and performance variability mitigation strategies for multiple workloads, including the N-body and Babelstream benchmarks, as well as the MiniFE mini-application, selected from HeCBench [10]. These workloads, implemented in both OpenMP and SYCL (using Intel's DPC++/C++ compiler [5]), were selected for their diverse computational and memory access characteristics, enabling a comprehensive analysis of performance variability across a range of workload types. The evaluation proceeds in two stages. First, the baseline performance variability is measured for each workload, implementation, and mitigation strategy with tracing enabled, 1000 times for each experiment. These measurements serve as a reference for analyzing the impact of noise injection in different configurations later. Although the tracing incurs some overhead, the measurements reported in Table 1 indicate that the performance degradation is below 1%. Therefore, tracing is not expected to meaningfully affect the results or the analysis presented in the following sections. Second, the noise injector is employed, configured with worst-case outliers identified from the baseline executions. This setup is used to compare the effectiveness of mitigation strategies across different workloads and programming models, providing insights into whether specific workload configurations can effectively mitigate approximate worst-case system noise, and to evaluate the injector's accuracy in reproducing system noise.

Tracing Overhead	Tracing Off	Tracing On	Increase
Nbody	0.450971154	0.453986513	0.67%
Babelstream	1.922135903	1.935881194	0.72%
MiniFE	1.06313158	1.065820493	0.25%

Table 1: Average execution time with tracing off and on.

Figures in this section illustrate sets of workload executions under distinct configuration labels, defined as follows:

- `Rm` (Roam): Threads are scheduled freely across all available CPU cores, without any affinity constraints.
- `TP` (Thread pinning): Each thread is pinned to a specific core, preventing thread migration during execution.
- `HK` (Housekeeping) and `HK2`: Housekeeping core configurations are implemented by reducing the amount of cores available to the workload. In the HK setup, 12.5% (`HK`) or 25% (`HK2`) of cores are left for background system tasks, restricting the workload to the remaining cores.
- `TPHK` and `TPHK2`: A combination of thread pinning and housekeeping.
- `RmHK` and `RmHK2`: A combination of roaming threads and housekeeping.

All tests were performed on two desktop platforms equipped with the following hardware and software configurations.

(1) AMD Ryzen 9950X3D: 16 physical cores, 32 logical cores (SMT enabled); Ubuntu 24.04.2 with Linux kernel version 6.11.0-25-generic; OneAPI DPC++/C++ compiler version 2025.1.0.20250317; GCC version 13.3.0.
(2) Intel i7 9700KF: 8 physical cores (no SMT), all set to a fixed clock speed of 4.7 GHz; Ubuntu 24.04 with Linux kernel version 6.8.0-51-generic; OneAPI DPC++/C++ compiler version 2025.0.4.20241205; GCC version 13.3.0.

We have conducted extensive experiments with the three workloads on both hardware platforms. Due to space constraints, we highlight representative results, focusing on key findings and trends that offer clear insights.

	Rm	RmHK	RmHK2	TP	TPHK	TPHK2
OMP	7.77	5.99	9.99	5.9	7.46	8.69
SYCL	7.18	7.84	5.55	6.75	7.63	5.36

Table 2: Average s.d. (ms) in baseline executions

5.1 Baseline performance evaluation

We start our evaluation by analyzing run-to-run performance variability of the selected workloads and assessing mitigation strategies without noise injection. The execution configurations, defined by workload type, programming model, and mitigation strategy, serve as baseline executions. The resulting traces are later used in Section 5.2 to characterize worst-case system behavior and average levels of inherent system noise. Table 2 reports average values across all evaluated benchmarks and application. Baseline results show that OpenMP and SYCL exhibit comparable levels of performance variability, measured as standard deviation (s.d.) in Table 2. However, OpenMP generally executes less time than SYCL on both platforms, suggesting a potential runtime advantage. In terms of mitigation strategies, housekeeping (`RmHK` or `RmHK2`) generally reduces performance variability for N-body. As expected, this comes at the cost of increased execution time, particularly for compute-intensive workloads such as N-body, due to a reduction in available computational resources. While housekeeping is less effective in this case, it proves more beneficial in worst-case scenarios with noise injection, where it suppresses performance outliers more effectively. We further elaborate on these cases in the following section.

Thread pinning shows modest benefits in reducing performance variability for OpenMP workloads. Unlike prior studies on typical HPC systems [7], the effect in our desktop-based setup is less pronounced. A potential reason is the difference in the system scale and noise characteristics between execution environments. To verify this, we re-executed these experiments under Linux runlevel 3, effectively disabling the graphical user interface to minimize GUI-induced noise. Although this generally reduced performance variability, overall trends remain unchanged. We speculate that the diminished effectiveness of thread pinning in our setup may stem from system scale differences. In particular, our tests were executed on single-socket desktop platforms, whereas prior work [7] was conducted on large-scale HPC system with multiple NUMA domains and up to 128 cores. In such systems, thread migration across NUMA domains can incur significant overhead, leading to increased interference and variability. In contrast, thread migrations within a single-socket desktop processor typically incur less overhead. Finally, combining thread pinning and housekeeping (`TPHK` and `TPHK2`) does not provide additional benefit. This suggests a limited synergy between the two strategies in our tested environments.

5.2 Performance evaluation under noise injection

In this section, we apply noise injection to emulate worst-case system behavior observed during baseline executions, and evaluate the effectiveness of different mitigation strategies and programming models in tolerating the injected noise. A total of ten worst-case traces are used in this evaluation, six collected from the Intel platform and four collected from the AMD platform. For each workload configuration, noise injection experiments are repeated 200 times to ensure statistical robustness.

	Rm	RmHK	RmHK2	TP	TPHK	TPHK2
N-body on Intel						
OMP #1	0.653	0.644	0.666	0.644	0.644	0.674
	45.5%	28.4%	15.0%	43.5%	27.5%	16.3%
SYCL #1	0.682	0.754	0.815	0.683	0.756	0.819
	13.3%	9.3%	6.1%	13.2%	9.4%	6.7%
OMP #2	0.562	0.518	0.588	0.556	0.529	0.593
	25.4%	3.2%	1.6%	23.8%	4.7%	2.2%
SYCL #2	0.661	0.703	0.773	0.665	0.705	0.774
	9.7%	1.9%	0.8%	10.1%	2.1%	1.0%
N-body on AMD						
OMP #1	1.392	0.832	0.902	1.398	0.784	0.884
	106.4%	10.0%	1.0%	107.2%	3.9%	-1.7%
OMP SMT #1	1.184	0.739	0.86	1.357	0.778	0.847
	69.6%	-0.1%	-5.5%	95.0%	3.4%	-1.5%
SYCL #1	1.056	0.947	1.033	1.193	0.943	1.015
	35.9%	3.8%	-0.6%	54.5%	4.0%	-1.1%
SYCL SMT #1	1.039	0.907	0.887	1.165	0.905	0.89
	18.6%	4.3%	-3.8%	34.0%	2.1%	-2.8%

Table 3: Average execution time (sec.) and the percentage increase (%) relative to the corresponding baseline configuration for N-body.

	Rm	RmHK	RmHK2	TP	TPHK	TPHK2
Babelstream on Intel						
OMP #1	1.951	1.916	1.897	1.915	1.892	1.879
	2.6%	0.1%	0.9%	1.1%	0.9%	1.2%
SYCL #1	2.175	2.147	2.134	2.177	2.15	2.142
	1.6%	-0.1%	1.2%	1.8%	0.3%	1.0%
OMP #2	2.452	1.918	1.894	2.372	2.086	1.985
	28.9%	0.2%	0.8%	25.2%	11.2%	6.9%
SYCL #2	2.403	2.242	2.173	2.415	2.269	2.205
	12.2%	4.3%	3.0%	12.9%	5.8%	4.0%
Babelstream on AMD						
OMP #1	1.004	0.905	0.888	1.016	0.893	0.881
	26.6%	15.8%	14.1%	28.7%	15.2%	14.1%
OMP SMT #1	1.013	0.9	0.876	1.016	0.91	0.893
	25.1%	10.1%	9.1%	26.2%	13.6%	12.4%
SYCL #1	1.111	1.067	1.047	1.126	1.074	1.053
	11.8%	8.1%	9.2%	13.4%	8.7%	10.2%
SYCL SMT #1	1.119	1.067	1.056	1.125	1.065	1.053
	10.6%	6.0%	8.1%	11.6%	6.2%	8.3%

Table 4: Average execution time (sec.) and the percentage increase (%) relative to the corresponding baseline configuration for Babelstream.

The results of the noise injection tests are presented in Tables 3, 4, and 5, showing each workload in its respective configuration. For each test, the tables report the average execution time and performance changes, calculated as the percentage difference between the baseline average and the observed test average. Numbers in the leftmost column of each table denote alternate noise injection configurations. Table 6 summarizes the results in all workloads for the OpenMP and SYCL implementations, reporting the average performance change across each configuration and benchmark. An ideal configuration would minimize both execution time and performance variability compared to other tested configurations under the same noise setting, indicating both high performance and

	Rm	RmHK	RmHK2	TP	TPHK	TPHK2
MiniFE on Intel						
OMP #1	1.243	1.24	1.239	1.246	1.611	1.772
	17.4%	17.0%	14.8%	18.2%	-2.1%	6.3%
SYCL #1	2.113	2.207	2.382	2.115	2.211	2.388
	5.3%	2.8%	1.6%	5.5%	3.1%	2.0%
OMP #2	2.128	1.99	1.891	2.211	2.774	2.468
	101.1%	87.7%	75.2%	109.9%	68.6%	48.0%
SYCL #2	2.774	2.696	2.874	2.77	2.704	2.873
	38.3%	25.5%	22.5%	38.2%	26.1%	22.7%
MiniFE on AMD						
OMP #1	0.874	0.882	0.859	0.864	1.092	1.106
	20.8%	12.0%	7.5%	22.3%	14.8%	14.0%
OMP SMT #1	0.934	0.921	0.92	0.932	1.168	1.166
	14.7%	5.6%	6.1%	18.8%	9.3%	8.0%
SYCL #1	1.63	1.65	1.709	1.615	1.644	1.707
	20.7%	18.3%	16.6%	20.6%	18.4%	17.6%
SYCL SMT #1	1.59	1.571	1.572	1.569	1.571	1.564
	16.6%	15.6%	15.7%	15.0%	15.3%	15.1%
OMP #2	1.228	1.236	1.286	1.378	2.081	2.095
	69.8%	56.9%	60.9%	95.0%	118.8%	116.1%
OMP SMT #2	1.188	1.214	1.212	1.405	2.123	2.125
	46.0%	39.1%	39.8%	79.2%	98.5%	96.8%
SYCL #2	2.07	1.925	1.971	2.04	1.939	1.99
	53.3%	38.0%	34.5%	52.3%	39.6%	37.1%
SYCL SMT #2	1.629	1.487	1.505	1.706	1.523	1.533
	19.5%	9.4%	10.8%	25.1%	11.8%	12.8%

Table 5: Average execution time (sec.) and the percentage increase (%) relative to the corresponding baseline configuration for MiniFE.

strong resistance to disturbance. Several trends can be identified from these results. As expected, employing housekeeping core(s) (e.g., `RmHK`, `RmHK2` and `TPHK2`) as an anomaly mitigation strategy proves effective in reducing worst-case performance degradation in most cases. Furthermore, in all tested cases, the use of housekeeping core(s) consistently reduces run-to-run performance variability. For larger-scale systems, where the benefit of additional cores diminishes in accordance to Amdahl's law, these findings indicate that reserving a portion of cores should be a general best practice when performance variability is a concern.

Regarding the effectiveness of thread pinning without housekeeping core(s) (`TP` column in Table 6), we do not observe any mitigation benefit compared to roaming threads (`Rm` column in Table 6). In most cases, the two configurations are comparable in both execution time and resilience to injected noise. Exceptions occur in the N-body and MiniFE tests with noise injection on the AMD system (Tables 3 and 5), where `Roam-omp` decently outperforms `TP-omp` relative to their respective baselines. Similar to the observations in Section 5.1 on the limited mitigation effectiveness of thread pinning, the equivalence of `TP` and `Rm` under noise injection observed here may be attributed to the relatively small core counts of the tested platforms and the explicit utilization of all cores by the workloads in `TP` and `Rm` experiments. On large-scale HPC systems with several CPU clusters, previous work has shown that thread pinning can be highly beneficial. However, because our experiments are conducted on single-CPU systems, migration overhead may be sufficiently low to be offset by the benefits of consistent workload thread progress.

When comparing the resilience of OpenMP and SYCL to injected noise, SYCL exhibits greater robustness, with an average improvement of 16.82% in Table 6 in noisy environments. However, several

	Rm	RmHK	RmHK2	TP	TPHK	TPHK2
OMP	42.85	20.43	17.24	49.58	27.73	24.22
SYCL	19.08	10.52	8.96	22.01	10.92	9.60

Table 6: Average relative performance change (%) under noise injection.

additional factors deserve consideration. First, both baseline and noise-injected executions using SYCL consistently yield overall longer raw execution time compared to OpenMP, which may limit its suitability to scenarios where performance resilience takes precedence over maximum execution speed Second, all but two of the noise configuration files, derived from worst-case traces, originate from OpenMP workload executions. These traces were selected because they present significant outliers, predominantly observed in OpenMP runs. This selection could potentially bias the generated noise configurations against OpenMP. Two further aspects may contribute to SYCL's higher resilience to injected noise: (1) its generally higher average execution time compared to OpenMP, and (2) the fixed active duration of noise injection, which matches the execution time of the emulated anomaly. However, given that the noise configurations generated from SYCL workloads produce results similar to those from OpenMP workloads, we have reasonable confidence that our findings are not substantially biased.

We also assess the accuracy of the proposed noise injector by measuring the relative difference in execution time between a recorded anomaly trace and the replicated run, expressed as the absolute value ($\left|\frac{Avg_{exec}}{Anomaly_{exec}} - 1\right|$), where Avg_{exec} is the workload's average execution time during noise injection and $Anomaly_{exec}$ is the execution time of the workload in the trace used to create the noise injection configuration. Lower values indicate higher replication accuracy. As shown in Table 7, the average accuracy across the evaluated workload configurations is 8.57%, a level regarded as acceptable for this study. Specifically, seven of the ten noise configurations used in the experiments yield average execution times within 8% of the worst-case trace execution time for equivalently configured workloads. The remaining three configurations exhibit larger deviations, ranging from 15.5% to 23%.

To validate the robustness of this evaluation, we conducted an additional test using a new worst-case trace. This test produced results that were considered compromised, as it results in all mitigation strategies performing with nearly identical execution times and yields the worst observed accuracy, at 25.74%. This issue originated

Benchmark	Config	Accuracy
N-body	**Rm-OMP**	3.80%
	TP-OMP	(-)2.40%
	Rm-SMT-OMP	6.47%
Babelstream	**Rm-OMP**	(-)0.10%
	TP-OMP	(-)15.50%
	TP-SYCL	6.99%
MiniFE	**Rm-OMP**	(-)7.30%
	TPHK2-OMP	18.60%
	TPHK-SMT-OMP	1.57%
	RmHK2-SYCL	22.95%

Table 7: Absolute accuracy of noise injection for each utilised worst-case trace.

in the final stage of noise configuration generation, where overlapping noise events on a single core were merged into a single noise event using a pessimistic assumption regarding the assigned scheduling policy. Consequently, large contiguous segments of diverse noise events were injected under the real-time scheduling policy. To address this issue, we further implemented an improved noise injection that avoided merging interrupt-based and thread-based noise, while also increasing the initial scheduling priority of thread-based noise events to encourage the scheduler to more aggressively schedule all noise assigned to the default policy. Evaluations with this approach not only resolved the earlier compromised runtime behavior for the above worst-case trace, from 25.74% to 5.70%, but also significantly improved the accuracy of the previously traces reported in Table 7, from 15.50% to 2.98%, and from 18.60% to 9.94%.

6 Discussion and Recommendation

This work primarily contributes a practical noise injection mechanism and an evaluation of multiple mitigation strategies for workloads implemented in two programming models. Based on the findings presented above, this section discusses broader implications, highlights limitations, and outlines recommendations for future research and deployment.

The effectiveness of each mitigation strategy depends heavily on workload characteristics, system environment, and execution configuration. Our key recommendations are as follows:

(1) **High-noise environments**: Housekeeping cores consistently improved performance across all workloads.
(2) **Memory-bound applications (e.g., Babelstream)**: Even under average noise, housekeeping cores yielded measurable gains.
(3) **Compute-bound applications (e.g., N-body)**: Under average-noise conditions, the best results were achieved without housekeeping cores; instead, thread pinning proved more effective.
(4) **General observation**: Leaving some cores, either logical or physical, unallocated by user workloads often reduced performance variability. In large-scale systems, the performance cost of unused cores is less critical due to Amdahl's law, making housekeeping particularly viable for parallel workloads.

There is no universal solution that works for all systems and requirements. Representative benchmarking of system conditions remains essential. We recommend reproducing the observed worst-case system noise via noise injection when evaluating workload configurations and mitigation strategies. Combining traditional benchmarking with noise injection allows testing under reproducible, diverse noise conditions rather than relying on stochastic occurrences. This approach helps developers balance average and worst-case performance, manage variability, deliver reliable output during optimization, and build confidence in the resilience of a workload to adverse background noise.

This work has some limitations. Only a limited set of benchmarks and mini-applications, i.e., N-body, Babelstream, and MiniFE, were evaluated, representing a narrow subset of application types. Noise injection was restricted to CPU occupation noise and did not include memory or I/O-related noise. Given the consistent accuracy for memory-bound benchmarks, we infer that the tested worst-case system noise conditions contained minimal memory activity. For systems with more significant memory-related noise, the noise injection methodology will need to be extended to emulate such effects. Additionally, experiments were conducted on consumer-grade hardware with a limited core count. On larger, multi-socket HPC systems, framework behavior and mitigation strategies could differ, due to increased thread migration overheads and altered system-level noise characteristics. For example, in this study, thread pinning and roaming threads often yielded comparable results in throughput and performance variability. Prior HPC-focused work, however, has shown that thread pinning can be substantially more beneficial in such environments.

7 Conclusion

In this work, we conducted reproducible performance evaluation of OpenMP and SYCL workloads under anomalous system conditions by proposing a noise injector capable of reproducibly emulating worst-case system noise. To reduce variability, we evaluated multiple mitigation strategies, such as thread pinning and housekeeping cores. The results show that controlled noise injection is a practical and effective method of assessing performance variability and identifying workload-specific mitigation approaches. Several promising directions remain for future work, such as extending the noise injector to capture a broader range of noise types, including I/O- and memory-related interference, validating results on larger-scale HPC systems, and improving accuracy by more precisely modeling scheduling policies and priorities during trace collection, and injection timing during noise injection. These enhancements could further improve the reproducibility of performance evaluation on modern multi-core systems. By enabling repeatable experimentation under controlled noisy conditions, this supports more reliable benchmarking, performance tuning, and the development of robust parallel applications.

Acknowledgments

This project has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland. Additionally, this work has received funding from the project PRIDE from the Swedish Foundation for Strategic Research with reference number CHI19-0048.

References

[1] A64FX. 2018. CTE-ARM. https://bsc.es/supportkc/docs/CTE-ARM/intro
[2] 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* (Kyoto, Japan) *(ICPP '19)*.
[3] Roberto Camacho Barranco and Patricia J. Teller. 2016. Analysis of the Execution Time Variation of OpenMP-based Applications on the Intel Xeon Phi. https://api.semanticscholar.org/CorpusID:1550490
[4] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. 2013. There goes the neighborhood: Performance degradation due to nearby jobs. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[5] Intel Corporation. 2024. *oneAPI DPC++ Compiler.* https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html Accessed: 2025-02-24.

[6] 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* (Denver, CO, USA) *(SC-W '23)*. 1614–1622.

[7] Minyu Cui and Miquel Pericas. 2025. Characterizing and Mitigating Performance Variability in Parallel Applications on Modern multicore Systems. In *Proceedings of the 22nd ACM International Conference on Computing Frontiers (CF '25)*. 151–158.

[8] Daniel Bristot de Oliveira, Daniel Casini, and Tommaso Cucinotta. 2023. Operating System Noise in the Linux Kernel. *IEEE Trans. Comput.* 72, 1 (2023), 196–207.

[9] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[10] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. doi:10.1109/ISPASS57527.2023.00041

[11] Beau Johnston, Jeffrey S. Vetter, and Josh Milthorpe. 2020. Evaluating the Performance and Portability of Contemporary SYCL Implementations. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 45–56.

[12] Joy Kitson Konstantinos Parasyris Harshitha Menon Isaac Minn Giorgis Georgakoudis Abhinav Bhatele Joshua H. Davis, Pranav Sivaraman. 2024. Taking GPU Programming Models to Task for Performance Portability. *Distributed, Parallel, and Cluster Computing (cs.DC); Performance (cs.PF)* (2024).

[13] Khronos Group. 2024. SYCL 2020 specification. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[14] Raimund Kirner and Peter Puschner. 2010. Time-Predictable Computing. In *Software Technologies for Embedded and Ubiquitous Systems*, Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–34.

[15] Edgar A. León, Ian Karlin, and Adam T. Moody. 2016. System Noise Revisited: Enabling Application Scalability and Reproducibility with SMT. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 596–607.

[16] LUMI. 2023. The low-noise mode on LUMI-G. https://lumi-supercomputer.github.io/LUMI-training-materials/User-Updates/Update-202308/lumig-lownoise/.

[17] Yehan Ma, Ruijie Fu, An Zou, Jing Li, Cailian Chen, Chenyang Lu, and Xinping Guan. 2024. Performance Optimization and Stability Guarantees for Multi-tier Real-Time Control Systems. In *2024 IEEE Real-Time Systems Symposium (RTSS)*. 187–200. doi:10.1109/RTSS62706.2024.00025

[18] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. 2017. An empirical survey of performance and energy efficiency variation on Intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing (E2SC'17)*.

[19] Aroon Nataraj, Alan Morris, Allen D. Malony, Matthew Sottile, and Pete Beckman. 2007. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12.

[20] OpenMP. 2018. OpenMP Application Programming Interface. https://www.openmp.org/spec-html/5.0/openmp.html

[21] Howard Porter Jr. Pritchard, Duncan Roweth, Dave Henseler, and Paul Cassella. 2012. Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. In *PROCEEDINGS OF THE CRAY USER GROUP, 2012.*

[22] D. Skinner and W. Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* 137–149.

[23] Hannes Weisbach, Balazs Gerofi, Brian Kocoloski, Hermann Härtig, and Yutaka Ishikawa. 2018. Hardware Performance Variation: A Comparative Study Using Lightweight Kernels. In *High Performance Computing*, Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis (Eds.). Springer International Publishing, Cham, 246–265.

[24] 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*. 1–16.