# Conquering Noise With Hardware Counters on HPC Systems

Marcus Ritter*, Ahmad Tarraf*, Alexander Geiß*, Nour Daoud†, Bernd Mohr†, Felix Wolf*

*Technical University of Darmstadt, Department of Computer Science, Germany
†Forschungszentrum Jülich GmbH, Jülich Supercomputing Centre, Germany
{marcus.ritter,ahmad.tarraf,alexander.geiss1,felix.wolf}@tu-darmstadt.de
{n.daoud,b.mohr}@fz-juelich.de

***Abstract*—With increasing system performance and complexity, it is becoming increasingly crucial to examine the scaling behavior of an application and thus determine performance bottlenecks at early stages. Unfortunately, modeling this trend is a challenging task in the presence of noise, as the measurements can become irreproducible and misleading, thus resulting in strong deviations from the actual behavior. While noise impacts the application runtime, it has little to no effect on some hardware counters like floating-point operations. However, selecting the appropriate counters for performance modeling demands some investigation. In this paper, we perform a noise analysis on various hardware counters. Using our noise generator, we add additional noise on top of the system noise to inspect the counters' variability. We perform the analysis on five systems with three applications in the presence of various noise patterns and categorize the counters across the systems according to their noise resilience.**

***Index Terms*—Hardware counters, performance analysis, noise, high-performance computing, parallel programming**

## I. Introduction

In the dawn of the Exascale systems, scientific applications, as well as the systems they are running on, are increasingly growing in performance and complexity. To gain a deeper understanding of the application behavior and to identify early performance bottlenecks, massive complex analyses are usually performed, which are often linked to a lot of effort, time, and costs. A much easier and still effective methodology to study the scaling behavior and identify application bottlenecks in an early stage is performance modeling, which has been wildly used in the HPC domain [1]–[3]. While performance modeling delivers good insight into the application behavior and its scalability behavior, a lot of factors can affect the models and thus the quality of the analysis. Besides factors linked to sudden changes in the application behavior (e.g., branch condition related to the number of ranks), other sources, such as noise in its variable forms and occurrences (see Section II), can also, influence the collected metrics and thus lead to inaccurate performance models. In fact, examining the performance of HPC applications in noisy environments has a long research history [4]–[7]. In general, noise can induce performance variability, which is the difference between execution times across repeated runs of an application in the same execution environment [8]. For example, performance variability can be a consequence of the variations in the execution environment, such as different process-to-node mappings [5] or thread-to-core mappings on NUMA systems [8]. Nevertheless, in spite of several attempts, performance variability is far from being eliminated and will remain an active research area as several studies have suggested [7]–[10].

One way to increase the accuracy of the performance models is to utilize hardware counters, as several approaches have shown [11], [12]. Power and energy estimation using hardware performance counters already has wider applicability in the HPC domain [13], [14]. While hardware counters exist on many systems, choosing the best ones to enhance performance models can be a challenging task. This becomes an even more demanding task in the presence of noise, as several of these counters are vulnerable to system noise. Traditionally, hardware events such as floating point operations (DP_OPS), as shown in Fig. 1, are known to be resilient against noise, as the short distributions in the presence (orange) as well as in the absence (blue) of induced noise show (for a detailed description, see Section III-F). On the other hand, considering

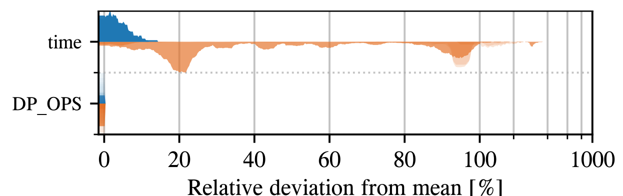


Fig. 1. Example showing the relative deviation from the arithmetic mean of the floating point operations (DP_OPS) and the runtime on the CM cluster for the three benchmarks examined in the paper.

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. Moreover, this work received funding by the Federal Ministry of Education and Research (BMBF), funding no. NHR2021HE, and the state of Hesse (HMWK), funding no. Kapitel 1502, Förderprodukt 19 NHR4CES as part of the NHR Program. We acknowledge the support of the European Commission and the German Federal Ministry of Education and Research (BMBF) under the EuroHPC Programmes DEEP-SEA (GA no. 955606, BMBF funding no. 16HPC015) and ADMIRE (GA no. 956748, BMBF funding no. 16HPC006K), which receive support from the European Union's Horizon 2020 programme and DE, FR, ES, GR, BE, SE, GB, CH (DEEP-SEA) or DE, FR, ES, IT, PL, SE (ADMIRE). Furthermore, this work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 449683531 (ExtraNoise project).

the time, variations exist in both cases, though they are by far larger in the presence of noise (orange) than in the absence (blue). But what about the remaining hardware events like branch/total instructions, TLB misses, instruction/data cache hits/misses, and many more? Is the observed behavior the same across different architectures? And which hardware events are suited for enhancing performance models in noisy environments? Furthermore, a question that has been already raised was: Can hardware performance counters be trusted [15]? Almost two decades later, considering how enhanced, complex, and huge modern HPC systems have become, there is a need to investigate the variability of hardware counters in the era of Exascale systems. Considering that only a few hardware events are deterministic [16], and that this varies across architectures, the resilience of the hardware counters to noise on the HPC systems needs to be examined. This will allow for the right selection of appropriate hardware events that not only boost performance models but also allow for the inclusion of robust metrics into these models, enhancing their noise resilience which directly leads to more accurate models. Hence, this paper aims to contribute by:

- Providing an analysis on the reliability of hardware counters. We analyze the hardware counters using PAPI [17] and examine all available preset events as well as a set of native events available on the five systems of different architectures (see Table I) using three benchmarks.
- Inspecting the variability of the counters by injecting to the already existing system noise additional noise using our noise generator NOIGENA to examine the counters' variability in the presence and absence of generated noise.
- Categorizing the counters across different systems and providing a user guide on which counters are suitable for enhancing performance models.

The remainder of this paper is structured as follows: In Section II, we discuss noise on HPC systems, look at related work, and introduce NOIGENA, the noise generator used in our analysis. In Section III, we elaborate on our methodology for identifying noise-resilient hardware counters, followed by presenting and discussing the results. Finally, we summarize our work in Section IV and state our future work.

## II. Noise on HPC System

Noise on HPC systems can have several sources, which range from intranode (node-level) sources up to internode (system-level) sources. In the following, we briefly describe the noise sources, followed by introducing our noise generator.

### A. Intranode Noise Sources

On the node level, Operating system (OS) noise represents a source of noise that manifests itself due to interrupts, polling, kernel threads, hypervisor, and many more. Operating systems can use, for example, timer interrupts to check on various activities periodically. Since each activity has a specific frequency and different amount of work that must be done, different timer interrupts can introduce different amounts of OS noise [18]. Due to its continuous nature, this type of OS noise can be classified as high-frequency continuous noise with a short duration. Moreover, the amount of noise is not negligible, i.e., OS noise can cause application performance degradation by a factor of two [4]. While the effect of OS noise is small on a single node (1-2% on average), it can be dramatic on large clusters, as the probability that at least one node out of thousands is slowed by some long kernel activity approaches 1 [19].

OS noise, caused by scheduling (e.g., the kernel swaps an HPC process out in order to run kernel daemons) and similar activities, can be classified as low-frequency long-duration noise [18]. This has been extensively researched in many studies [4], [19], [20]. Variations in the working conditions (operating temperature variations, power capping, etc.) can be another source of intranode noise. While power capping, for example, is a desired feature to limit the power consumption, it results in CPU frequency variation, which impacts the application execution time and causes performance inhomogeneity [21]. Aside from the working conditions, hardware variations (e.g., aging and manufacturing variability) can also be an intranode noise source. Not only are the hardware components affected by manufacturing variability [21]–[23] due to the fabrication process at the beginning of their life span, but also by hardware aging during their life. As the behavior of the transistors deviates from the original intended behavior during the chip lifetime, the chip suffers from performance degradation and will consequently fail to meet some of its specifications [24].

Another interesting source of intranode noise is shared resource contention. Due to the shared resources on a node like processors, memory, and network, resource contention can even occur on the node level. In case many background I/O threads are used to carry out I/O operations, and the application itself issues several threads to take advantage of multi-core CPUs, oversubscription can occur if the combined number of threads is larger than the number of available hardware threads. This causes interference and leads not only to resource competition between the background I/O threads and application threads but also to competition among the I/O threads themselves [25].

### B. Internode Noise Sources

Considering internode noise, several well-known sources include network contention, parallel file system (PFS) contention, system-level power capping, load imbalance, and many more. Looking at the same example again, if asynchronous I/O threads are operating at the same time as the application communication, network contention will occur not only slowing down the application but also the background I/O threads, hence, prolonging the impact on other resources (CPU, memory bandwidth, etc.) as well [25]. Other noise sources, such as PFS contention and how to reduce it, are a research field of their own [26]–[28]. Moreover, the interference caused by resource contention often occurs by design, as resource sharing is expected to improve system utilization [8]. Noise sources, such as PFS contention, can be classified

as a high-impact source with irregular patterns. As examining all sources of noise would by far exceed the scope of this paper, we limit the scope to noise generated by the OS and from the shared resources leaving the I/O for future work.

### C. Generating Noisy Measurements

To examine the variability and robustness of the hardware counters under intensive noise, we generate noise in addition to the noise that is already present in the system. Thus, we can examine the hardware counters in the presence as well as in the absence of the injected noise. For that, we use our newly developed tool called NOIGENA (NOIse GENerator Application), which will be made publicly available soon after code restructuring. NOIGENA is composed of several benchmarks that are run consecutively in a configurable pattern and time frame in parallel to the underlying tested application aiming at stressing it with three noise types resulting from shared resources contention, which are: memory, network, and I/O. NOIGENA is composed of Stream[1], FzjLinkTest[2], and IOR[3] benchmarks responsible for memory, network, and I/O noise, respectively. For this paper, we will only focus on memory and network noise patterns.

The Stream benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. By configuring the size of the memory, Stream acts as a memory noise source. The FzjLinkTest program is a parallel ping-pong test between all possible MPI connections of a machine. By configuring the number of processes to use, the message size, and whether to send in serial or parallel and other configurations, FzjLinkTest acts as a communication noise source. To control the duration and the type of noise created by NOIGENA, we use patterns that determine for how long and with which configuration the benchmarks are executed while NOIGENA is running. Listing 1 shows such a configuration pattern. A pattern is defined by a sequence of noise types each run for a given amount of time with no noise periods intervening in between to simulate high and low-frequency noise patterns. Thus, NOIGENA enables us to simulate the different types of noise an application is exposed to during its runtime in a real-life scenario. A sequence of noise can be repeated either infinitely, for a given amount of time, or randomly. The left pattern in Listing 1 provides an example of a sequence that repeats until NOIGENA is terminated, alternately generating two seconds of network noise followed by two seconds without noise, two seconds of memory noise, and two seconds without noise. The pattern on the right generates random noise for a fixed duration.

## III. Identifying Noise-Resilient Counters

Finding hardware events across different systems that are noise resilient demands deep investigations. Even in strictly

[1]https://www.cs.virginia.edu/stream/

[2]https://www.fz-juelich.de/en/ias/jsc/services/user-support/jsc-software-tools/linktest

[3]https://sourceforge.net/projects/ior-sio/files/IOR%20latest/IOR-2.10.3/

```
PATTERN_1:
  Sequence:
    - REPEATED_NOISE:
        REPEAT: inf
        Sequence:
          - NETWORK_NOISE: 2
          - NO_NOISE: 2
          - MEMORY_NOISE: 2
          - NO_NOISE: 2
```

```
PATTERN_2:
  Sequence:
    - RANDOM_NOISE:
        Seed: 53
        TIME: 1000
        NETWORK_NOISE: 50
        NO_NOISE: 10
        MEMORY_NOISE: 40
```

Listing 1. Exemplary configuration patterns used by NOIGENA to determine the type and the duration (in seconds) of the generated noise. The listing on the left shows an itself endlessly repeating noise pattern, while the one on the right generates random noise for a given amount of time.

controlled environments, run-to-run variations can occur [16]. Hence, determining hardware events that are deterministic across different systems is a challenging task. This task becomes even more difficult when noise is induced into the system. This lies in the fact that major sources of non-determinism are linked to the operating system's activities, context switching, hardware interrupts, the performance overhead of measurements, and the precision of the measuring tools [29]. Unlike other work in this field, [16], our primary focus is neither to find deterministic events nor to identifying causes for hardware counter deviations, but rather to identify hardware counters that are suitable for enhancing performance models and thus for examining the scaling behavior of HPC applications. Moreover, we are interested in finding hardware counters that are to some extent robust against OS noise and resource contentions, thus resulting in a small deviation between run-to-run simulations on the same system. Additionally, we aim to classify hardware counters across different systems in regard to their variability and thus usefulness for performance modeling.

### A. PAPI and Hardware Counters

We use PAPI (Performance Application Programming Interface) [30] to access the various performance counters on the different systems. PAPI provides an interface to hardware performance counters across different architectures through two kinds of events: native and preset events. While native events are platform-dependent, preset events are platform neutral and are mapped to single or linear combinations of the native countable events. To access the counters, PAPI provides a low-level as well as a high-level API. Through the years, PAPI has evolved from monitoring CPU-related events to also include network cards, graphics accelerator cards, parallel file systems, and more [31]. Though performance counters find a wide range of applicability, including performance optimization, simulator validation and power or temperature estimation, they are also coupled with a few limitations, including accuracy, overhead, and determinism [32]. Regardless of which counter usage mode is employed (statistical sampling or aggregating event counts), inaccuracy can happen in both, as a results of the overhead of the counter interfaces, the cache pollution it causes, the lack of hardware support for precisely identifying an event's address, or even from advanced features modern chips utilize [32]. However, in this paper, we are not concerned

with determinism, which is the situation where two identical runs of the same program result in the same results of the monitored events but rather with the robustness of the counters against noise.

To identify noise-resilient counters, we conducted an extensive analysis investigating all preset events available on the used evaluation systems, such as the total instructions, floating point operations, or the total cycles. The Figs. 3-13 in Section III-F outline the results of our analysis and provide a complete list of all preset events that we analyzed. As of PAPI 6.0.0.1, 108 preset events exist, though the number of available events varies on the examined systems listed in Table I. Going from the top to the bottom order of the evaluation systems from Table I, the number of available preset events are: 58, 58, 17, 21, and 43. Considering the native events, their number varies tremendously between the systems. Hence, we selected a small subset of native counters, such as the PERF hardware CPU cycles and the PERF hardware instructions, that are potentially useful for performance analysis. However, due to the accessibility of preset events across various systems, we are focused on the preset events and limited the analysis to at most 18 PAPI native events, which are outlined in Section III-F.

TABLE I
LIST OF ALL SYSTEMS AND THEIR HARDWARE USED FOR THE ANALYSIS.

| Alias | Name | System hardware |
|---|---|---|
| Jureca | JURECA DC Module, std. compute nodes | 480 nodes, 2x AMD EPYC 7742 CPUs (64 cores), 512 GB DDR4 RAM (3 200 MHz), InfiniBand HDR100 (100 GBit/s) |
| ESB | DEEP-EST, Extreme Scale Booster | 75 nodes, 1x Intel Xeon Cascade Lake Silver 4215 CPU (8 cores, 16 threads), 48 GB DDR4 RAM (2 400 MHz), InfiniBand EDR (100 GBit/s) |
| CM | DEEP-EST, Cluster Module | 50 nodes, 2x Intel Xeon Skylake Gold 6146 CPUs (12 cores, 24 threads), 192 GB DDR4 RAM (2 666 MHz), InfiniBand EDR (100 GBit/s) |
| Jetson | OACISS, Franken-cluster Jetson ARM64 | 12x Jetson Tegra TX1, Quad-Core ARM Cortex®-A57 MPCore, 4 GB 64-bit LPDDR4 RAM, 1 GBit/s ethernet |
| Cyclops | OACISS, Franken-cluster Cyclops | 2x 20c IBM Power9 CPUs (40 cores, 160 threads), 384 GB of RAM, BNX2 10G Ethernet NICs, 2x Infiniband EDR (25 GBit/s) |

### B. Performing Noise Analysis on Hardware Counters

To decide whether a hardware counter is noise-resilient or not, we investigate how the measured counters' values change when repeating the measurements using the exact same experiment configuration while being exposed to different levels of noise. The experiment configuration consists of the used applications' configuration parameters such as the problem size and the system configuration, i.e., the resource allocation. For our analysis, we repeated each experiment configuration at least five times to observe how the counters' values change due to different amounts and types of noise. To gather the measurements, we used PAPI and Score-P [33] to automatically instrument all kernels of our application benchmarks using compiler wrapping and then measured the counters' values for each of them.

We run NOIGENA simultaneously with our application benchmarks using a repeating noise sequence to generate the noise. By doing so it creates a pre-configured amount of noise until it automatically terminates together with the target application. To emulate the conditions on a large-scale HPC system, we employ different noise patterns to simulate low and high-frequency noise, but also to distinguish between noise caused by network congestion or memory effects. Additionally, we used different noise patterns for each experiment repetition to maximize the impact of noise on the application measurements and their divergence between repetitions. Another possible source of inaccuracy is multiplexing. The operating system uses multiplexing if there are not enough physical hardware counters available for the requested counters, which might result in a certain amount of inaccuracy in the results [32]. To avoid this and to inspect each counter on its own, we only measure one counter at a time during each experiment run.

To analyze whether the values of a counter change when a performance experiment with the same configuration is repeated, we calculate the relative deviation from the measured arithmetic mean value of the counter in percent $(|v_i - \bar{v}|)/\bar{v} \cdot 100\%$ for all instrumented application kernels individually. This metric provides a good measure of how much a counter's values scatter without noise and when exposed to noise. Since we use Score-P, we automatically obtain the measured counters' values and the runtime for each instrumented application kernel per process (MPI rank), OpenMP thread, and experiment repetition. To calculate the relative deviation from the arithmetic mean counter value, we first add up the counter values $u_i$ of all threads per MPI rank and application kernel and then calculate their arithmetic mean $\bar{u}$. This is necessary as the work done by each thread can vary vastly, which would lead to huge divergences when calculating the arithmetic mean for all values without this step. Next, we calculate the arithmetic mean counter value $\bar{v}$ for each individual application kernel and experiment configuration for all its processes and repetitions using the previously calculated set of arithmetic means $\bar{u}_i$. Finally, we use the previously calculated arithmetic means $\bar{u}_i = v_i$ and compare them to $\bar{v}$ to calculate the relative deviation for all ranks and their repetitions in percent using the following equation $(|v_i - \bar{v}|)/\bar{v} \cdot 100\%$.

### C. Application Benchmarks

For our analysis, we investigated the performance of three HPC applications: LULESH, MiniFE, and LAMMPS. LULESH is a proxy app created by the Lawrence Livermore National Laboratory to solve a simple Sedov blast problem with analytic answers—but represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications [34]. For our analysis, we use LULESH 2.0 with OpenMP and MPI support.

MiniFE (also known as HPCCG) is a mini-app that mimics the finite element generation, assembly, and solution for an unstructured grid problem, which is required by many engineering applications [35]. It is not intended to be a true physics problem but sufficiently realistic for performance

analysis, for example, for studying the scalability of competing systems. For our analysis, we used the optimized OpenMP code `openmp-opt` with MPI support.

LAMMPS is a classical molecular dynamics code focusing on materials modeling [36]. It has potentials for modeling a large number of physics problems such as solid-state materials (metals, semiconductors), to model atoms, or more generically, as a parallel particle simulator [36]. For benchmarking purposes, LAMMPS features five standard problems that are widely used for performance analysis. For our experiments, we use the atomic fluid, Lennard-Jones (LJ) potential with 2.5 sigma cutoff (55 neighbors per atom), and NVE integration with OpenMP and MPI support as described on the LAMMPS benchmarks page (https://www.lammps.org/bench.html).

### *D. System and Software Configurations*

For our analysis, we conducted performance measurements on five different systems ranging from large-scale HPC clusters to the small-scale portable ARM platform. Today's systems are diverse and feature a variety of different CPU architectures as well as memory, storage solutions, and accelerators. To thoroughly investigate the counters noise-resilience, we selected four of today's most commonly used computing platforms for our evaluation which are: Intel, AMD, IBM, and ARM. Table I lists all systems and their hardware configuration that were used for the analysis. Due to the different number of resources each system provides, we adjusted the experiment configurations accordingly, as shown in Table II.

To compile the application benchmarks and conduct the measurements, we used PAPI 6.0.0.1 and OpenMP on all systems. In addition, we used the Intel compiler, ParaStation MPI, and Score-P 6.0 for ESB and CM. For Jureca, we used GCC, ParaStation MPI, and Score-P 7.1. Finally, for the remaining systems, we used GCC, OpenMPI, and Score-P 7.1.

### *E. Experiment Configurations*

To simulate the efficient use of the evaluation systems, we parallelized our applications and NOIGENA using both OpenMP and MPI. For NOIGENA, we additionally map the used CPU cores such that the execution of the LinkTest and Stream benchmarks are intervening with the application's execution. Furthermore, we used five different infinite noise patterns with different sequences of noise types and durations, such as the one shown in Listing 1. As described in SectionIII-B, we repeat the measurements for each counter and experiment configuration five times. To increase the amount of divergence in the measured performance metrics among the repetitions and simulate the conditions that prevail on a large-scale HPC system, we created five different noise patterns, one for each repetition. The following patterns were used for our analysis. Pattern one: no noise 2 sec, network noise 2 sec, no noise 2 sec, memory noise 2 sec. Pattern two: no noise 1 sec, network noise 3 sec, no noise 1 sec, memory noise 3 sec. Pattern three: network noise 4 sec, memory noise 4 sec. Pattern four: network noise 8 sec, memory noise 8 sec. Pattern five: no noise 6 sec, network noise 12 sec, memory noise 8 sec. For the simulation without induced noise, we simply repeated the measurements five times with the same experimental setup.

Besides the NOIGENA settings, an experiment configuration is determined by the application parameters such as the problem size $s$, and the resource allocation, i.e., the number of computational nodes $n$, the number of MPI processes $p$, and the number of OpenMP threads $t$. Table II shows the experiment configurations for each application and system that we used for the analysis. To investigate the impact of the resource allocation and the problem size on the counter values, we conducted experiments using various different values for $n$ and $s$. Furthermore, we measured each experiment twice, once with noise and once without. This enables us to compare the counters' values divergence in both scenarios with each other and the runtime values, providing important insights for performance analysis. Hence, with $k$ preset hardware counters, we have performed $k \times \text{len}(n) \times \text{len}(p) \times 2 \times 5$ experiments on each system per app adding up to a total of 26950 for the preset events only.

TABLE II
EXPERIMENT CONFIGURATIONS USED FOR THE ANALYSIS.

| App | System | Experiment configuration |
|---|---|---|
| MiniFE | CM | $n = [1, 2, 4, 8]$, $p = n$, $t = 12p$, $s = 20^3p$ |
| | ESB | $n = [1, 2, 4, 8, 12, 16, 32]$, $p = n$, $t = 8p$, $s = 50^3p$ |
| | Jureca | $n = [4, 8, 12, 16, 20, 24]$, $p = n$, $t = 128p$, $s = 20^3p$ |
| | Jetson | $n = [2, 3, 4, 5, 6]$, $p = n$ , $t = 2p$, $s = 50^3p$ |
| | Cyclops | $n = 1$, $p = [4, 8, 12, 16, 20]$, $t = 4p$, $s \approx 24^3p$ |
| LULESH | CM | $n = [1, 8, 27]$, $p = n$, $t = 12p$, $s = 10^3p$ |
| | ESB | $n = [1, 8, 27]$, $p = n$, $t = 8p$, $s = 30^3p$ |
| | Jureca | $n = [1, 8, 27]$, $p = n$, $t = 128p$, $s = 10^3p$ |
| | Jetson | $n = 8$, $p = n$, $t = 2p$, $s = 15^3p$ |
| | Cyclops | $n = 1$, $p = [1, 8, 27, 64]$, $t = p$, $s = 5^3p$ |
| LAMMPS | CM | $n = [1, 2, 4, 8]$, $p = n$, $t = 12p$, $s = 20^3p$ |
| | ESB | $n = [1, 2, 4, 8, 12, 16, 32]$, $p = n$, $t = 8p$, $s = 20^3p$ |
| | Jureca | $n = [1, 2, 4, 8, 16]$, $p = n$, $t = 128p$, $s = 20^3p$ |
| | Jetson | $n = [2, 3, 4, 5, 6]$, $p = n$, $t = 2p$, $s = 20^3p$ |
| | Cyclops | $n = 1$, $p = [4, 8, 12, 16, 20]$, $t = 4p$, $s = 20^3p$ |

### *F. Evaluation of the Results and Plot Anatomy*

The evaluation results are displayed as relative population distributions for the relative deviation $((|v_i - \bar{v}|)/\bar{v} \cdot 100\%)$ for each PAPI preset counter. For each counter, the maximum value of the counter was used to scale the peak occurrence of the relative deviation. Fig. 2 shows an example of such a

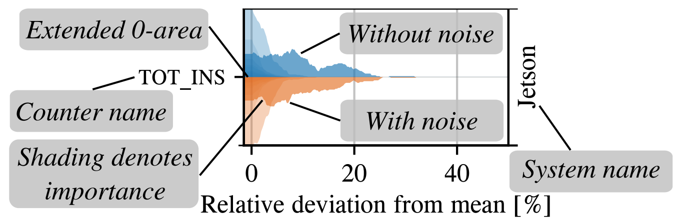


Fig. 2. Anatomy of the relative deviation graphs.

plot. We provide both the results with and without injected noise (orange lower part and blue upper part, respectively)

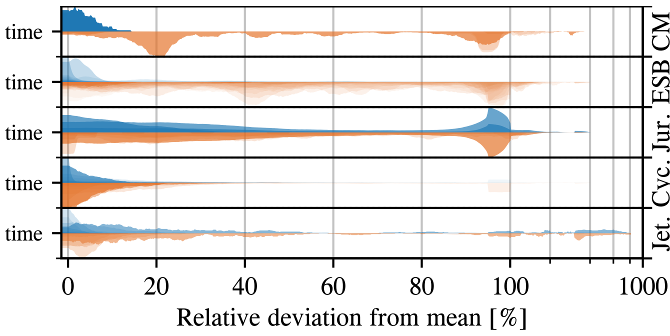


Fig. 3. Distribution plot showing the runtime variations from the arithmetic mean of the five systems with the three benchmarks from Section III-E. For figure explanation see Section III-F.

as adjoined distribution plots allowing for direct comparison. For improved visibility, we extended the values at 0% relative deviation into the negative area. Note that we only show callpaths that resulted in more than 1% of the total counter value. The distribution plots are grouped by similar deviation behavior and functional unit.

Intuitively, the height of each distribution plot describes how often the corresponding relative deviation occurs, and the intensity (alpha) shows the importance of the region to the overall behavior. We define the importance as the callpath's share of the total counted value across all applications per counter because large values indicate the parts of the application that exhibit more of the behavior described by that counter. Therefore, the heights of a callpath are scaled with the share of the total counted values to emphasize callpaths with bigger counter values. The layering of the distribution plots allows us to represent the overall distribution behavior while retaining characteristic individual behavior. As a point of reference, we treated the runtime as a hardware counter and displayed the distributions in Fig. 3 to show the effects caused by the injected noise.

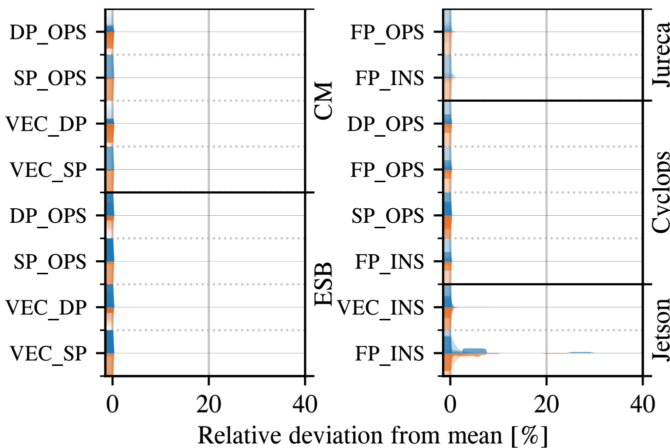


Fig. 4. Noise analysis for the floating point counters across the five systems with three benchmarks. As expected, these counters are noise resilient.

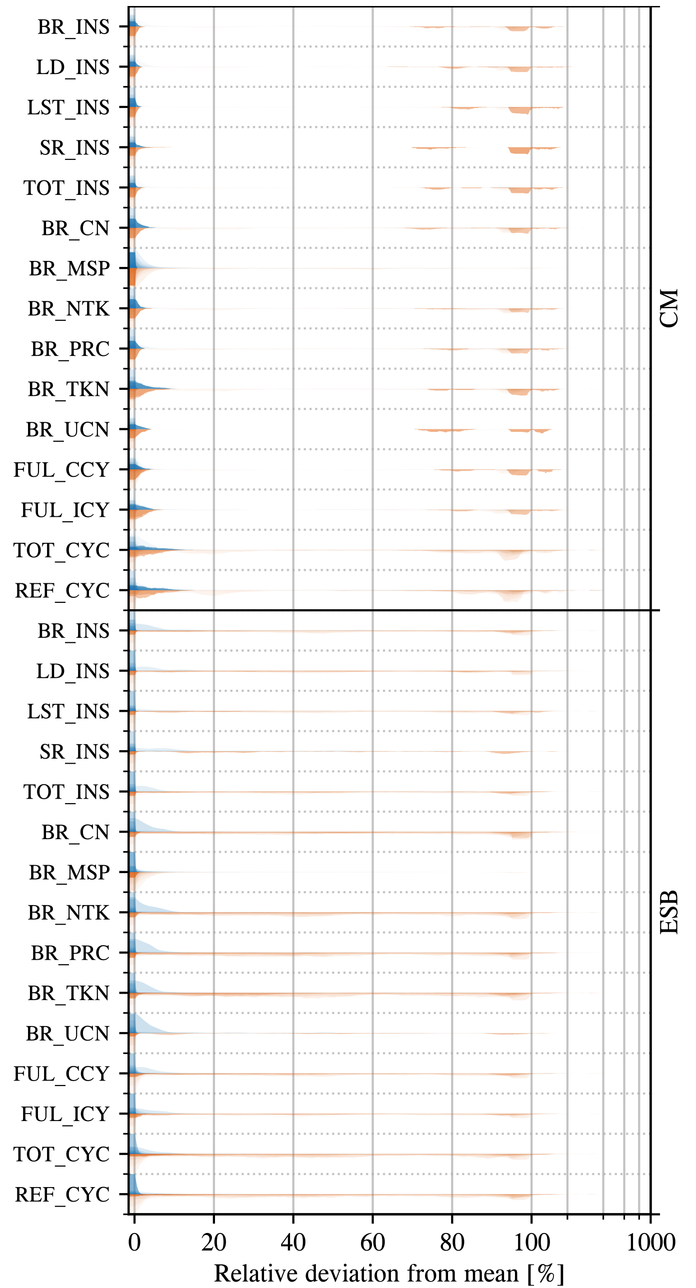


Fig. 5. Distribution plot outlining the relative deviations from the arithmetic mean of the instructions and cycles for ESB and CN.

### G. Discussion and Best Practice Guide

We classified the preset hardware events across the five systems into categories according to their variations: floating-point counters (Fig. 4), Intel instruction and cycle counters (Fig. 5), ARM and IBM instruction and cycle counters (Fig. 6), stall cycles and AMD instructions (Fig. 7), instruction cache counters (Fig. 8), L2 (Fig. 9) and L3 (Fig. 10) data cache counters, L2/L3 total cache counters (Fig. 11), L3 cache counters for IBM (Fig. 12), unclassified counters (Fig. 13)

For the selected native events, we observed a similar behavior to the corresponding preset events. However, as the number

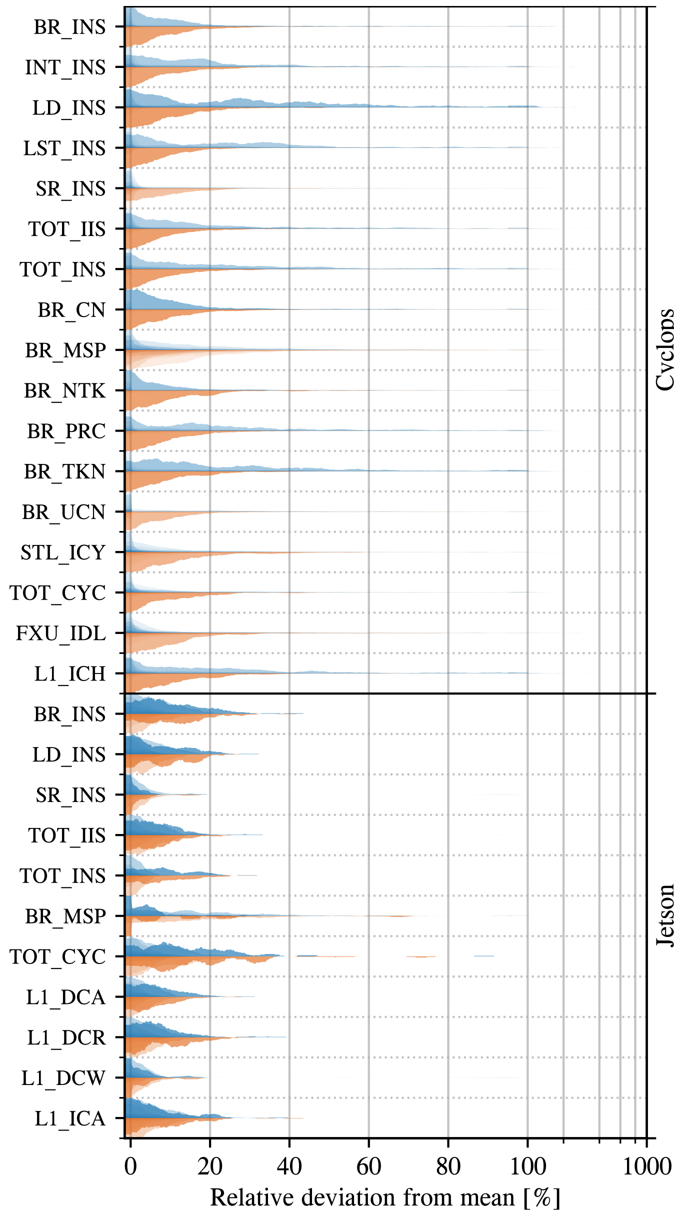


Fig. 6. Distribution plot outlining the relative deviations from the arithmetic mean of the instructions and cycles for Cyclops and Jetson.

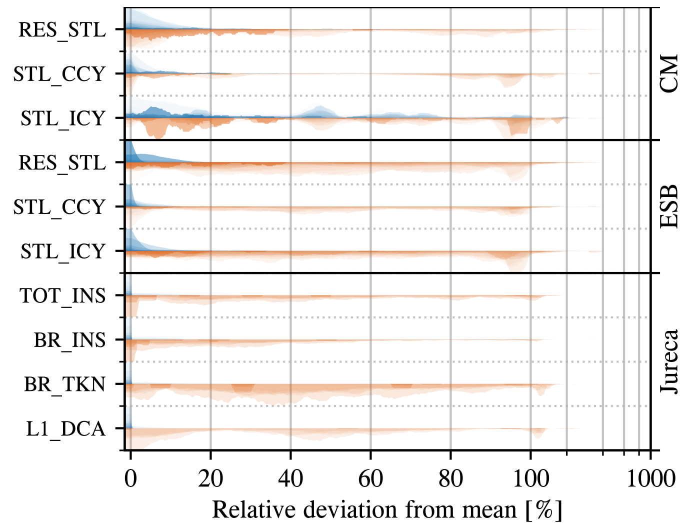


Fig. 7. Distribution plot outlining the relative deviations from the arithmetic mean for all benchmarks of the stall and reference cycles for CM, ESB, and the instructions for Jureca.

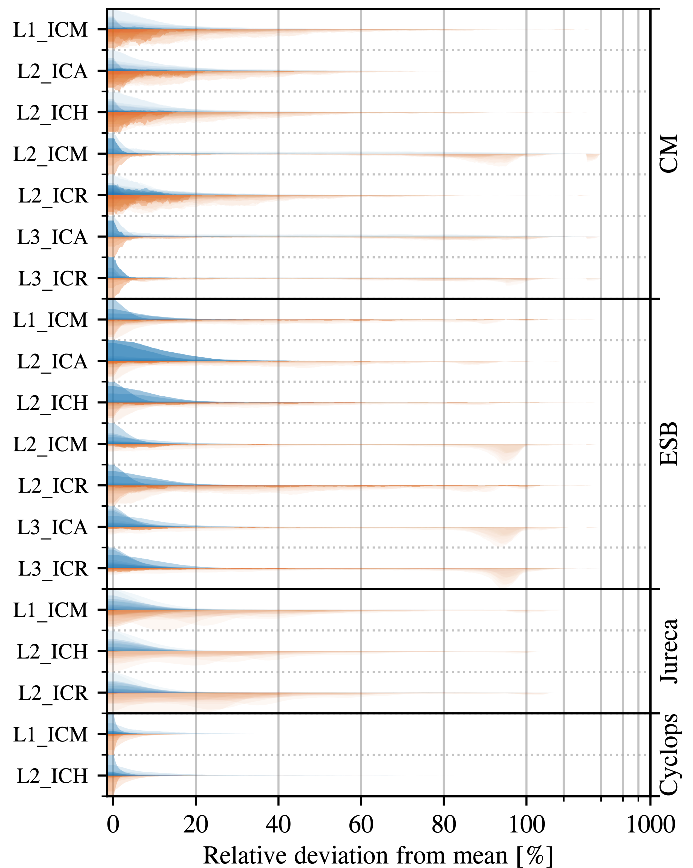


Fig. 8. Distribution plot showing the relative deviations from the arithmetic mean of the instruction cache accesses/misses for CM, ESB, Jureca, and Cyclops.

and type of traced events were too different on the systems (e.g., 18 on the Intel systems, 5 on Cyclops, and 9 on Jureca), we left out the evaluation figure.

We could confirm that floating-point counters are noise resilient. Moreover, we have observed that several hardware events reveal smaller deviations than the time, making them suitable for performance modeling, though they exhibit larger variation compared to floating-point operations. Also, several metrics reveal similar behavior, as the cache hits and misses across the cache levels. For example, the L1 cache misses behave similarly to the L2 cache accesses. Analogously, the same goes for L2 cache misses and L3 cache accesses. Furthermore, we have seen that the performance counters are differently affected by noise. We gathered our findings in the

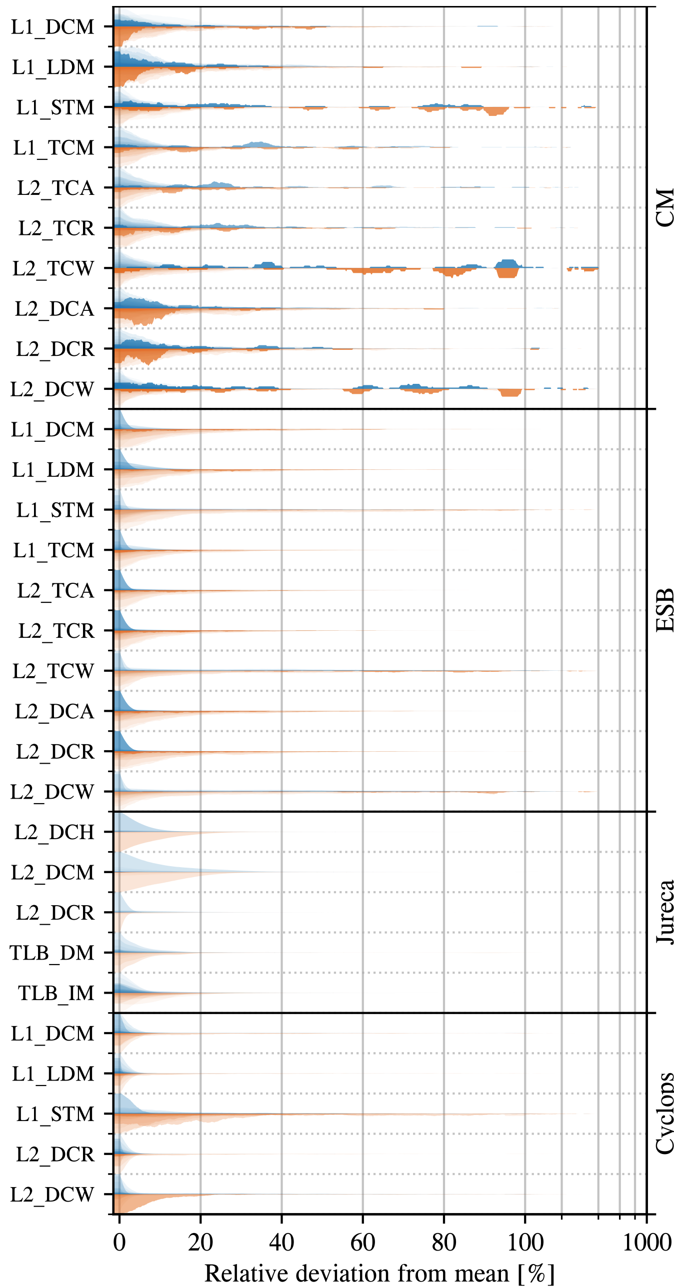


Fig. 9. Distribution plot illustrating the relative deviations from the arithmetic mean of the L1/L2 cache accesses/misses for CM, ESB, Jureca, and Cyclops.

best practice guide provided below.

**Best Practice User Guide**: Under the influence of noise and independent of the system architecture, the best hardware counters for enhancing performance models are all counters measuring either floating point operations or instructions such as the DP_OPS, FP_INS, or VEP_SP shown in 4.

**Intel**: The instructions and cycles from Fig. 5 reveal a strong deviation (95%) for the less important callpaths (low intensity) and a small deviation for important callpath (high intensity). Hence, we recommend to still use them, as compared to the time (Fig. 3). Their variation is by far smaller making them

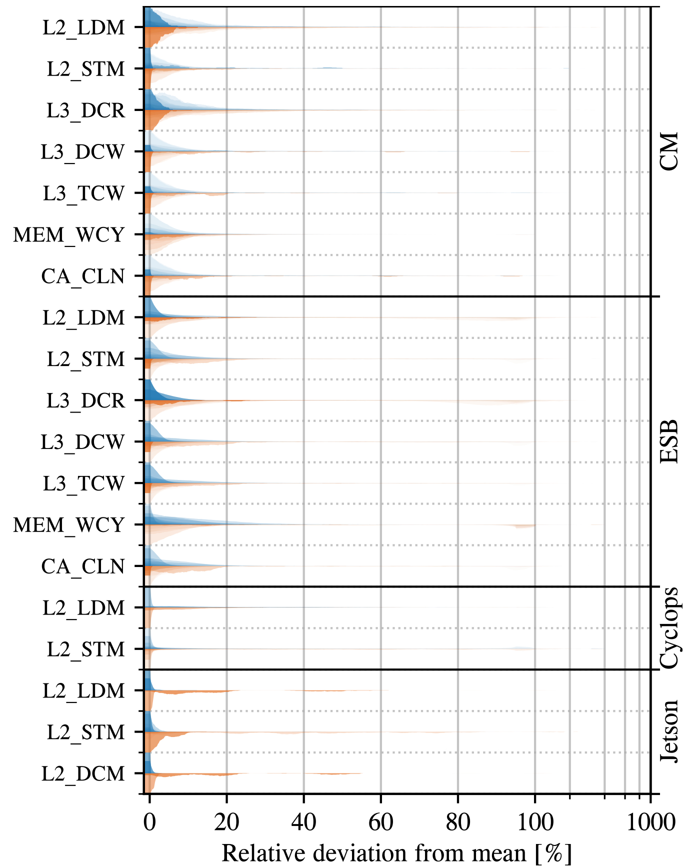


Fig. 10. Distribution plot showing the relative deviations from the arithmetic mean of the L2/L3 data cache accesses/misses for CM, ESB, Cyclops, and Jetson.

suitable for performance modeling. We recommend avoiding stall and reference cycles (Fig. 7) as the deviations are relative large. While the L3 counters as well as counters like TLB_DM are accurate in the absence of noise, they are strongly affected by it and start to variate significantly (Fig. 11). Thus they should be avoided for performance modeling.

**AMD**: The total and branch instruction show a strong deviation (Fig. 7) when subjected to noise, but still less than the time. The L2 cache access and misses, TLB_DM and TLB_IM show a small deviation (at most 20% divergence).

**IBM**: In general, for significant callpaths, the deviation is acceptable ($< 30\%$), but still there is a small number of high deviations (see Fig. 6). However, instructions such as BR_TKN, BR_PRC, LD_INS, and LST_INS should be avoided for performance modeling as the deviations are quite large. The same goes for L3 related counters (see Fig. 12).

**ARM**: As Fig. 6 shows, the instruction and cycles counters reveal small deviations. Excepted for the TOT_CYC, we would recommend all for performance modeling.

Counters measuring instruction cache-related values as the ones shown in Fig. 8 are strongly affected by noise on most systems. A similar behavior is observed in the L1/L2 data cache counters from Fig. 9. In contrast, L2/L3 data cache counters are more robust against noise (see Fig. 10).

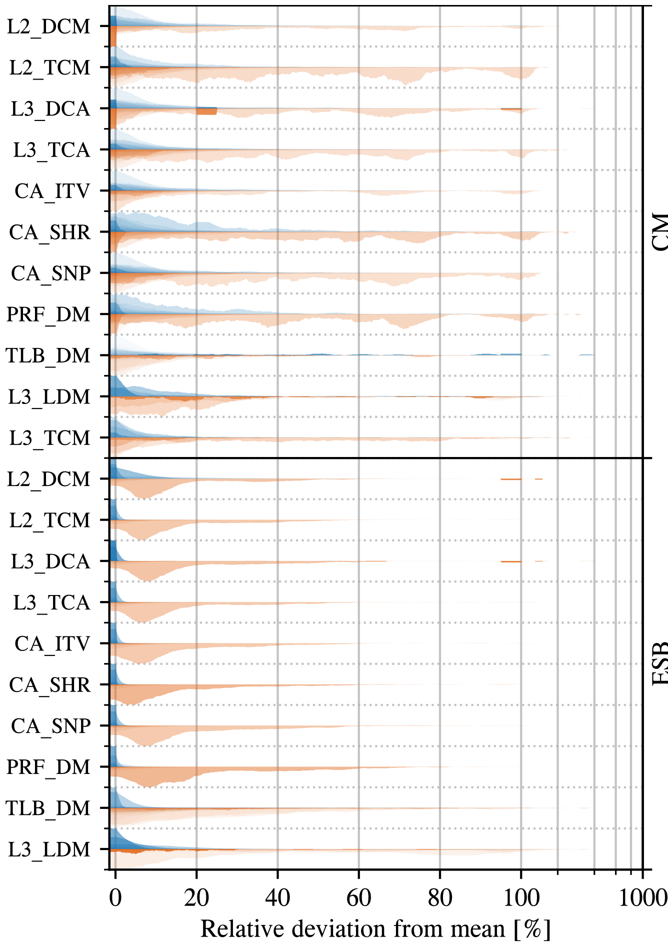


Fig. 11. Distribution plot outlining the relative deviations from the arithmetic mean of the L2/L3 cache accesses, misses, and requests for CM and ESB.

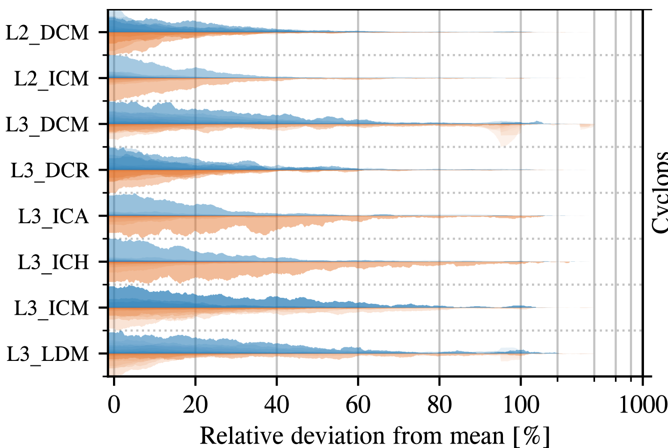


Fig. 12. Distribution plot showing the relative deviations from the arithmetic mean of the L2/L3 cache accesses/misses on Cyclops.

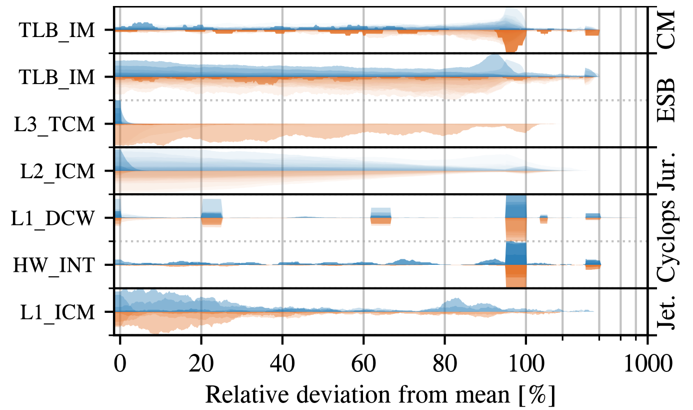


Fig. 13. Distribution plot outlining the relative deviations from the arithmetic mean of counters that could not be categorized for CM, ESB, Jureca, Cyclops, and Jetson.

## IV. Conclusion

In this paper, we investigated the noise-resilience of hardware counters using three application benchmarks and five evaluation systems with diverse hardware architectures. We examined all available PAPI preset events and a selected set of native events on these systems and analyzed their reliability in the presence and absence of injected noise. Our analysis confirmed the results of previous studies, showcasing that all counters measuring either floating point operations or instructions are noise-resilient. Overall, it unveiled that, independent of the system architecture, noise generally affects hardware counters. Furthermore, the reliability of many counters depends significantly on the system architecture. While the instruction and cycle counters are highly reliable on CM and ESB, on Cyclops, Jetson, and especially Jureca, they are much more prone to be influenced by noise. Therefore, our best practice user guide enables application developers and researchers aiming to analyze or optimize the performance of their code to easily identify the hardware counters relevant for performance analysis for their system architecture. Future work will focus on generating performance models with the inspected hardware counters and expand the noise analysis to include other noise sources, such as I/O contention.

## Acknowledgements


The authors gratefully acknowledge the computing time provided to them on the high-performance computer Lichtenberg at the NHR Centers NHR4CES at TU Darmstadt and the JURECA supercomputer at Jülich Supercomputing Centre (JSC). Special thanks to Sameer Shende for providing the authors with access to the clusters of the University of Oregon.


## References


[1] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, Nov. 2011, pp. 1–12.

[2] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, “Using automated performance modeling to find scalability bugs in complex codes,” in *Proc. of the international conference on high performance computing, networking, storage and analysis*, 2013, p. 45.

[3] H. Stengel, J. Treibig, G. Hager, and G. Wellein, “Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model,” in *Proc. of the 29th ACM on International Conference on Supercomputing*, ser. ICS ’15. New York, NY, USA: Association for Computing Machinery, Jun 2015, p. 207–216. [Online]. Available: https://doi.org/10.1145/2751205.2751240

[4] F. Petrini, D. Kerbyson, and S. Pakin, “The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q,” in *SC ’03: Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, Nov. 2003, pp. 55–55.

[5] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, “There goes the neighborhood: Performance degradation due to nearby jobs,” in *SC ’13: Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2013, p. 1–12.

[6] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener, “Understanding performance interference in next-generation HPC systems,” in *SC ’16: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, p. 384–395.

[7] T. Patki, J. J. Thiagarajan, A. Ayala, and T. Z. Islam, “Performance optimality or reproducibility: that is the question,” in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, Nov 2019, p. 1–30.

[8] D. A. Nikitenko, F. Wolf, B. Mohr, T. Hoefler, K. S. Stefanov, V. V. Voevodin, A. S. Antonov, and A. Calotoiu, “Influence of noisy environments on behavior of HPC applications,” *Lobachevskii Journal of Mathematics*, vol. 42, no. 7, pp. 1560–1570, Jul. 2021.

[9] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, “Run-to-run variability on Xeon Phi based Cray XC systems,” in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’17. New York, NY, USA: Association for Computing Machinery, Nov 2017, p. 1–13. [Online]. Available: https://doi.org/10.1145/3126908.3126926

[10] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, “Taming performance variability,” in *Proc. of the 13th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI’18. USA: USENIX Association, 2018, p. 409–425.

[11] N. Ding, S. Xu, Z. Song, B. Zhang, J. Li, and Z. Zheng, “Using hardware counter-based performance model to diagnose scaling issues of HPC applications,” *Neural Computing and Applications*, vol. 31, no. 5, pp. 1563–1575, May 2019.

[12] N. Ding, V. W. Lee, W. Xue, and W. Zheng, “APMT: an automatic hardware counter-based performance modeling tool for HPC applications,” *CCF Transactions on High Performance Computing*, vol. 2, no. 2, p. 135–148, Jun 2020.

[13] K. O’Brien, I. Pietri, R. Manumachu, A. Lastovetsky, and R. Sakellariou, “A survey of power and energy predictive models in HPC systems and applications,” *ACM Computing Surveys*, vol. 50, Oct 2017.

[14] G. Tsafack Chetsa, L. Lefèvre, J. Pierson, P. Stolf, and G. Da Costa, “Exploiting performance counters to predict and improve energy performance of HPC systems,” *Future Generation Computer Systems*, vol. 36, p. 287–298, Jul 2014.

[15] V. M. Weaver and S. A. McKee, “Can hardware performance counters be trusted?” in *2008 IEEE International Symposium on Workload Characterization*, Sep 2008, p. 141–150.

[16] V. M. Weaver, D. Terpstra, and S. Moore, “Non-determinism and overcount on modern hardware performance counter implementations,” in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, TX, USA: IEEE, Apr 2013, p. 215–224.

[17] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting performance data with PAPI-C,” in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer, 2010, p. 157–173.

[18] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, “A quantitative analysis of OS noise,” in *2011 IEEE International Parallel & Distributed Processing Symposium*. Anchorage, AK, USA: IEEE, May 2011, p. 852–863.

[19] R. Gioiosa, S. A. McKee, and M. Valero, “Designing OS for HPC applications: Scheduling,” in *2010 IEEE International Conference on Cluster Computing*. Heraklion, Greece: IEEE, Sep 2010, p. 78–87. [Online]. Available: http://ieeexplore.ieee.org/document/5600318/

[20] K. B. Ferreira, P. Bridges, and R. Brightwell, “Characterizing application sensitivity to os interference using kernel-level noise injection,” in *SC ’08: Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, Nov 2008, p. 1–12.

[21] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi, “Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing,” in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin Texas: ACM, Nov. 2015, pp. 1–12.

[22] S. Borkar, “Designing reliable systems from unreliable components: The challenges of transistor variability and degradation,” *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov. 2005.

[23] S. Dighe, S. R. Vangal, P. Aseron, S. Kumar, T. Jacob, K. A. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. K. De, and S. Borkar, “Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core TeraFLOPS processor,” *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 184–193, Jan. 2011.

[24] D. Kraak, M. Taouil, S. Hamdioui, P. Weckx, F. Catthoor, A. Chatterjee, A. Singh, H.-J. Wunderlich, and N. Karimi, “Device aging: A reliability and security concern,” in *2018 IEEE 23rd European Test Symposium (ETS)*. Bremen: IEEE, May 2018, pp. 1–10.

[25] S.-M. Tseng, B. Nicolae, F. Cappello, and A. Chandramowlishwaran, “Demystifying asynchronous I/O Interference in HPC applications,” *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 391–412, Jul. 2021.

[26] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, “On the root causes of cross-application I/O interference in HPC storage systems,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, p. 750–759.

[27] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, “Scheduling the I/O of HPC applications under congestion,” in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, p. 1013–1022.

[28] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, “GekkoFS — a temporary burst buffer file system for HPC applications,” *Journal of Computer Science and Technology*, vol. 35, no. 1, p. 72–91, Jan 2020.

[29] M. Mushtaq, P. Benoit, and U. Farooq, “Challenges of using performance counters in security against side-channel leakage,” in *CYBER 2020 - 5th International Conference on Cyber-Technologies and Cyber-Systems*, Nice, France, Oct 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02979362

[30] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting performance data with PAPI-C,” in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[31] F. Winkler, “Redesigning PAPI’s high-level API,” University of Tennessee, Tech. Rep. ICL-UT-20–03, 2020.

[32] S. Moore, D. Terpstra, and V. Weaver, *Software Interfaces to Hardware Counters*, ser. Chapman & Hall/CRC Computational Science. CRC Press, Nov 2010, vol. 20102662, p. 33–48. [Online]. Available: http://www.crcnetbase.com/doi/abs/10.1201/b10509-4

[33] A. Knüpfer, C. Rössel, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel *et al.*, “Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir,” in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[34] I. Karlin, J. Keasler, and R. Neely, “LULESH 2.0 updates and changes,” Tech. Rep. LLNL-TR-641973, August 2013.

[35] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, “Improving performance via mini-applications,” Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[36] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in’t Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen *et al.*, “LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales,” *Computer Physics Communications*, vol. 271, p. 108171, 2022.