

# A Quantitative Analysis of OS Noise

Alessandro Morari<sup>\*</sup>, Roberto Gioiosa<sup>\*</sup>, Robert W. Wisniewski<sup>§</sup>, Francisco J. Cazorla<sup>\*†</sup>, Mateo Valero<sup>\*†</sup>

<sup>\*</sup>Computer Science Division  
Barcelona Supercomputing Center, ES

<sup>†</sup>Computer Architecture Department  
Universitat Politècnica de Catalunya, ES

<sup>‡</sup>IIA-CSIC (Spanish National Research Council)

<sup>§</sup>IBM T. J. Watson Research Center

{alessandro.morari,roberto.gioiosa,francisco.cazorla}@bsc.es, bobww@watson.ibm.com, mateo@ac.upc.edu

**Abstract**—Operating system noise is a well-known problem that may limit application scalability on large-scale machines, significantly reducing their performance. Though the problem is well studied, much of the previous work has been qualitative.

We have developed a technique to provide a *quantitative* descriptive analysis for each OS event that contributes to OS noise. The mechanism allows us to detail all sources of OS noise through precise kernel instrumentation and provides frequency and duration analysis for each event. Such a description gives OS developers better guidance for reducing OS noise. We integrated this data with a trace visualizer allowing quicker and more intuitive understanding of the data.

Specifically, the contributions of this paper are three-fold. First, we describe a methodology whereby detailed quantitative information may be obtained for each OS noise event. Though not the thrust of the paper, we show how we implemented that methodology by augmenting LTTng. We validate our approach by comparing it to other well-known standard techniques to analyze OS noise. Second, we provide a case study in which we use our methodology to analyze the OS noise when running benchmarks from the LLNL Sequoia applications. Our experiments enrich and expand previous results with our quantitative characterization. Third, we describe how a detailed characterization permits to disambiguate noise signatures of qualitatively similar events, allowing developers to address the true cause of each noise event.

## I. INTRODUCTION

Operating system (OS) noise (or jitter) is a well-known problem in the High Performance Computing (HPC) community. Petrini et al. [1], [2] showed how OS noise may limit application's scalability, severely reducing performance on large scale machines and large system activities (e.g., network file system server). Later studies [2], [3] confirmed the early conclusions on different systems and identified timer interrupts and process preemption as main sources of OS noise [4], [5]. Most of those studies are “qualitative” in that they do a good job characterizing the overall effect of OS noise on the scalability of parallel applications, but tend not to identify and characterize each kernel noise event. For example, many studies showed that the timer interrupt is an important

source of OS noise, but few of them provided information about which activities are executed during each timer interrupt. Such information would be helpful to developers trying to reduce OS noise. For example, operating systems use timer interrupts to periodically start several kinds of activities, such as bookkeeping, watchdogs, page reclaiming, scheduling, or drivers' bottom halves. Each activity has a specific frequency and its duration varies according to the amount of work to process, and thus different timer interrupts can introduce different amounts of OS noise. Our detailed quantitative analysis allows developers to differentiate and characterize the noise induced by each event triggered on an interrupt and thus address the pertinent sources.

Lightweight and micro kernels are common approaches taken to reduce OS noise on HPC systems. Many supercomputers have run a lightweight kernel, such as Compute Node Kernel (CNK) [6] or Catamount [7]. Lightweight kernels provide functionality for a targeted set of applications (for example, HPC or embedded applications) and usually introduce negligible noise. They usually do not take periodic timer interrupts or TLB misses, but typically provide restricted scheduling and dynamic memory capability.

The lightweight approach worked well in the past when HPC applications only required scientific libraries and a message passing interface (MPI) implementation [8], [9]. The NAS benchmark suite [10] contains examples of these kind of applications. Also, the requirements of many other of the largest applications, such as previous Gordon Bell winners [11], [12], most of the Sequoia benchmark suite [13], and others [14], [15] can be satisfied by this approach.

However, modern HPC applications are becoming more complex, such as UMT from the LLNL Sequoia benchmarks [13], Climate code [16], and UQ [17]. Moreover, in order to improve performance there is a growing interest in richer shared-memory programming models, e.g., OpenMP [18], PGAS, such as UPC, Charm++, etc. At the same time, dynamic libraries, python scripts, dynamic memory allocation,

checkpoint-restart systems, tracers to collect performance and debugging information, and virtualization [19], are seeing wider use in HPC systems. All these technologies require richer support from the OS and the system software. Moreover, applications being run on supercomputers are no longer coming strictly from classical scientific domains but from new fields, such as financial, data analytics, and recognition, mining, and synthesis (RMS), etc. Current trends indicate large petascale to exascale systems are going to desire a richer system software ecosystem.

Possible solutions to support such complexity include: 1) extending lightweight and micro kernels to provide support for modern applications needs as mentioned above; 2) tailoring a general purpose OS, such as Linux or Solaris, to the HPC domain (e.g., the Cray Linux Environment for Cray XT machines); and 3) running a general purpose OS in a virtualized environment (e.g., Palacios [19]). All the three scenarios above have the potential to induce more noise than yesterday's lightweight kernels. Thus, it will become more important to be able to accurately identify and characterize noise induced by the system software.

We have developed a technique to provide a quantitative descriptive analysis for each event of OS noise. The mechanism allows us to detail all sources of OS noise through precise kernel instrumentation and provides frequency and duration analysis for each event. In addition to providing a much richer set of data, we have integrated this collected data with the Paraver [20] visualization tool allowing developers to more easily analyze the data and get an intuitive sense for its implications. Paraver is a classical tool for parallel application's performance analysis, and integrated with our data can provide a view of the OS noise introduced throughout the execution of HPC applications.

Overall, this paper makes three main contributions. First, we describe a methodology whereby detailed quantitative information may be obtained for each OS noise event. As mentioned our methodology is most useful for HPC OS designers and kernel developers trying to provide a system well suited to run HPC applications. Though not the thrust of the paper, we show how we implemented that methodology by augmenting LTTng. We validate our approach by comparing it to other well-known standard techniques to analyze OS noise, such as FTQ (Fixed Time Quantum). Second, we provide a case study in which we use our methodology to analyze the OS noise when running benchmarks from the LLNL Sequoia applications. Although our methodology and tools may well be useful for application programmers, in this work we focus on providing the insights that can be gained at the system level. Thus, while we do run real applications, it is not for the purpose of studying those applications, but rather for demonstrating that our methodology allows us to obtain a better understanding of the system while running real applications. Our experiments enrich and expand previous results with our quantitative characterization. Third, we describe how this detailed characterization allows us to disambiguate noise signatures of qualitatively similar events allowing developers

to address the true cause of a given noise event.

The structure of the paper is as follows. Section II describes related work. Section III describes our mechanism to obtain a detailed quantitative description for each OS noise event. Section IV shows the results of applying our technique to analyze benchmarks from the LLNL Sequoia applications. Section V provides two examples of how to use our mechanism to disambiguate noise signatures. We conclude in Section VI.

## II. RELATED WORK

Operating system noise and its impact on parallel applications has been extensively studied via various techniques, including noise injection [2]. In the HPC community, Petrini et al. [1] explained how OS noise and other system activities, not necessarily at the OS level, could dramatically impact the performance of a large cluster. In the same work, they observed that the impact of the system noise when scaling on 8K processors was large due to *noise resonance* and that leaving one processor idle to take care of the system activities led to a performance improvement of  $1.87\times$ . Though the authors did not identify each source of OS noise, a following paper [4] identified timer interrupts, and the activities started by the paired interrupt handler, as the main source of OS noise (*micro OS noise*). Nataraj et al. [21], also describe a technology to measure kernel events and make this information available to application-level performance measurements tools.

Others [5], [22] found the same result using different methodologies. The work of De et al. [22] is the most similar to ours. They perform an analysis of the OS noise instrumenting the functions `do_IRQ` and `schedule`, and obtaining interrupts and process preemptions statistics. Our approach is similar, but we instrumented all kernel entry points and activities, thus providing a complete OS noise analysis. As a consequence, we are able to measure events like page faults and softirqs that significantly contribute to OS noise.

The other major cause of OS noise is the scheduler. The kernel can swap HPC processes out in order to run other processes, including kernel daemons. This problem has also been extensively studied [1], [4], [5], [22]–[24], and several solutions are available [24], [25].

Studies group OS noise into two categories of high-frequency, short-duration noise (e.g. timer interrupts) and low-frequency, long-duration noise (e.g. kernel threads) [2]. Impact on HPC applications is higher when the OS noise resonates with the application, so that high-frequency, fine-grained noise affects more fine-grained applications, and low-frequency, coarse-grained noise affects more coarse-grained applications [1], [2].

The impact of the operating system on classical MPI operations, such as collective, is examined in Beckman et al. [26].

There are several examples of operating systems in the literature designed for HPC applications. Solutions can be divided into three classes: micro kernels, lightweight kernels (LWKs), and monolithic kernels. *L4* [27] is a family of micro-kernels designed and updated to achieve high independence from the platform and improve security, isolation, and robustness. The

*Exokernel* [28]–[30] was developed with the idea that the OS should act as an executive for small programs provided by the application software, so the Exokernel only guarantees that these programs use the hardware safely. *K42* [31] is a high performance, open source, general purpose research operating system kernel for cache-coherent multiprocessors that was designed to scale to hundreds of processors and to address the reliability and fault-tolerance requirements of large commercial applications. Sandia National Laboratories have also developed LWKs with predictable performance and scalability as major goals for HPC systems [7].

IBM *Compute Node Kernel* for Blue Gene supercomputers [6], [32] is an example of a lightweight OS targeted for HPC systems. CNK is a standard open-source, vendor-supported, OS that provides maximum performance and scales to hundreds of thousands of nodes. CNK is a lightweight kernel that provides only the services required by HPC applications with a focus of providing maximum performance. Besides CNK, which runs on the compute nodes, Blue Gene solutions use other operating systems. The I/O nodes, which use the same processors as the compute nodes, run a modified version of Linux that includes a special kernel daemon (*CIOD*) that handle I/O operations, and front-end and service nodes that run a classical Linux OS [32].

Lightweight and micro kernels usually partner with library services, often in user space, to perform traditional system functionality, either for design or performance reasons. CNK maps hardware into the user’s address space allowing services such as messaging to occur, for efficiency reasons, in user space. The disadvantages of CNK come from its specialized implementation. It provides a limited number of processes/threads, minimal dynamic memory support, and contains no `fork/exec` support [6]. Moreira et al. [32] show two cases (socket I/O and support for Multiple Program Multiple Data (MPMD) applications) where IBM had to extend CNK in order to satisfy the requirements of a particular customer. Overall, experience shows that this situations are quite common with micro and lightweight kernels.

There are several variants of full weight kernels that have customized general-purpose OSes. ZeptoOS [33]–[35] is an alternative, Linux-based OS, available for Blue Gene machines. ZeptoOS aims to provide the performance of CNK while providing increased Linux compatibility. Jones et al. [23] provides a detailed explanation of the modification introduced to IBM AIX to reduce the execution time of `MPI Allreduce`. They showed that some of the OS activities were easy to remove while others required AIX modifications. The authors prioritize HPC tasks over user and kernel daemons by playing with the process priority and alternating periods of favored and unfavored priority. HPL [24] reduces OS noise introduced by the scheduler by prioritizing HPC processes over user and kernel daemons and by avoid unnecessary CPU migrations.

Shmueli et al. [3] provide a comparison between CNK and Linux on BlueGene/L, showing that one of the main items limiting Linux scalability on BlueGene/L is the high number of TLB misses. Although the authors do not address schedul-

ing, by using the HugeTLB library, they achieve scalability comparable to CNK (although not with the same performance).

Mann and Mittal [36] use the secondary hardware thread of IBM POWER5 and POWER6 processors to handle OS noise introduced by Linux. They reduce OS noise by employing Linux features such as the real time scheduler, process priority, and interrupt redirection. The OS noise reduction comes at the cost of losing the computing power of the second hardware thread. Mann and Mittal consider SMT interference a source of OS noise.

### III. MEASURING OS NOISE

The usual way to measure OS noise on a compute node consists of running micro benchmarks with a known and highly predictable amount of computation per time interval (or quantum), and measuring the divergence in each interval from the amount of time taken to perform that computation. An example of such as technique is the *Finite Time Quantum* (FTQ) benchmark proposed by Sottile and Minnich [37]. FTQ measures the amount of work done in a fixed time quantum in terms of *basic operations*. In each time interval  $T$ , the benchmark tracks how many basic operations were performed. Let  $N_{max}$  be the maximum number of basic operations that can be performed in a time interval  $T$ . Then we can indirectly estimate the amount of OS noise, in terms of basic operations, from the difference  $N_{max} - N_i$ , where  $N_i$  is the number of basic operations performed during the  $i$ -th interval.

Figure 1a shows the output of FTQ on our test machine. Each spike represents the amount of time the machine was running kernel code instead of FTQ (obtained multiplying the number of missing basic operations by the time required to perform a basic operation). Whether the kernel interrupted FTQ one, two, or more times, and what the kernel did during each interruption is not reported.<sup>1</sup> This imprecision is one disadvantage of indirect external approaches to measuring OS noise. An advantage of this approach is that experiments are simple and provide quick relative comparisons between different versions as developers work on reducing noise.

As we stated earlier, our goal was to provide a more detailed *quantitative* evaluation of each OS noise event. In order to obtain this information in Linux, instrumentation points includes all the kernel entry and exit points (interrupts, system calls, exceptions, etc.), and the main OS functions (such as the scheduler, `softirqs`, or memory management [39]). To obtain a detailed trace of OS events, we extended the Linux Trace Toolkit Next Generation (LTTng) [40], [41] with 1) an infrastructure to analyze and covert the LTTng output into formats useful to analyze OS noise, and 2) extra instrumentation points inside the Linux kernel. We call this extended LTTng infrastructure LTTNG-NOISE.

The output from LTTNG-NOISE includes a *Synthetic OS Noise Chart* and an *OS Noise Trace*. The former is a graph similar to the one generated by FTQ and provides a view of

<sup>1</sup>It is possible to guess that the small, frequent spikes are related to the timer interrupt, as was pointed out by previous work [4], [5], [38].

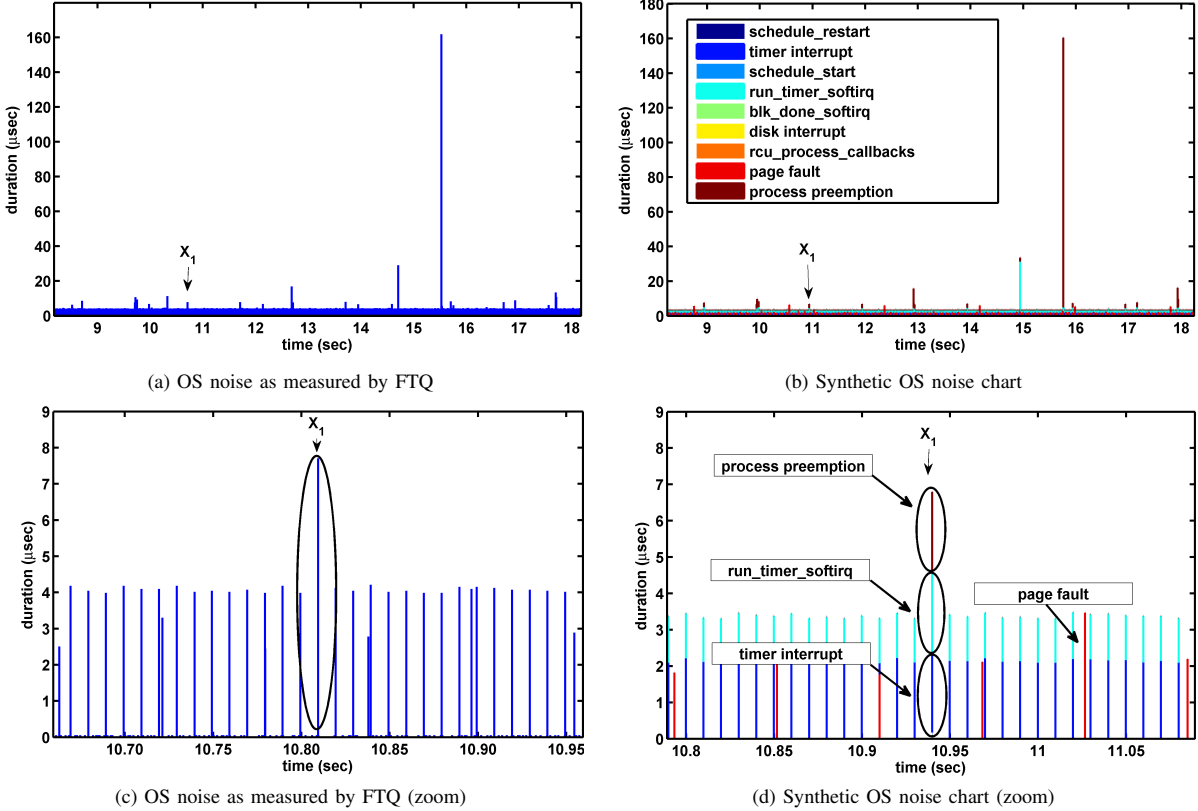


Fig. 1: Measuring OS noise using FTQ. Figures 1a and 1c report the direct OS overhead obtained multiplying the execution time of a basic operation by the number of missing operations. Figures 1b and 1d report time as measured by LTTNG-NOISE.

the amount of noise introduced by the OS. The latter is an execution trace of the application that shows all kernel activities. Figure 1b shows the Synthetic OS Noise Chart for FTQ generated by LTTNG-NOISE (as shown in Section V, we can obtain similar graphs for any application). The OS Noise Trace can be visualized with standard trace visualizers commonly used for HPC application performance analysis. The current implementation of LTTNG-NOISE supports the Paraver [20] trace format, but other formats can be generated relatively easily by performing a different offline transformation of the original trace file.

#### A. LTTNG-NOISE

A concern about LTTNG-NOISE was the overhead introduced by the instrumentation. If the overhead is too large, the instrumentation may change the characteristics of the applications making the analysis invalid. Minimizing introduced noise, or more accurately keeping it below a measureable level, was especially critical for us as we were specifically trying to measure noise. In addition, HPC applications are susceptible to this induced noise causing an additive detrimental effect. Fortunately, LTTng was successfully designed with similar goals, and our kernel modifications do not add extra overhead. In particular, LTTng has been designed with the following properties: 1) low system overhead, 2) high-scalability for multi-core architectures, and 3) high-precision. Low system

overhead is obtained with a pre-processing approach, i.e., the kernel is instrumented statically and data are analyzed offline. High-scalability is ensured by the use of per-cpu data and by employing lock-less tracing mechanisms [40]. Finally, high-precision is obtained using the CPU timestamp counter providing a time granularity on the order of nanoseconds. The results of the low-overhead design of LTTng and our careful modification is an overhead in the order of 0.28% (average among all the LLNL Sequoia applications we tested).

Since it is not possible to know in advance which kernel activities affect HPC applications, we collected all possible information. LTTng already provides a wide coverage of the Linux kernel. Even with these capability, it is still important to determine which events are needed for the noise analysis and, eventually add new trace points. To this end, we located all the entry and exit points and identified the components and section of code of interest. We then modified LTTng by adding extra information to existing trace points and new instrumentation points, for those components that were not already instrumented.

Another important consideration is which kernel activities should be considered noise (i.e., timer interrupts) and which are, instead, services required by applications and should be considered part of the normal application's execution (i.e., a read system call). In this paper, we consider OS noise

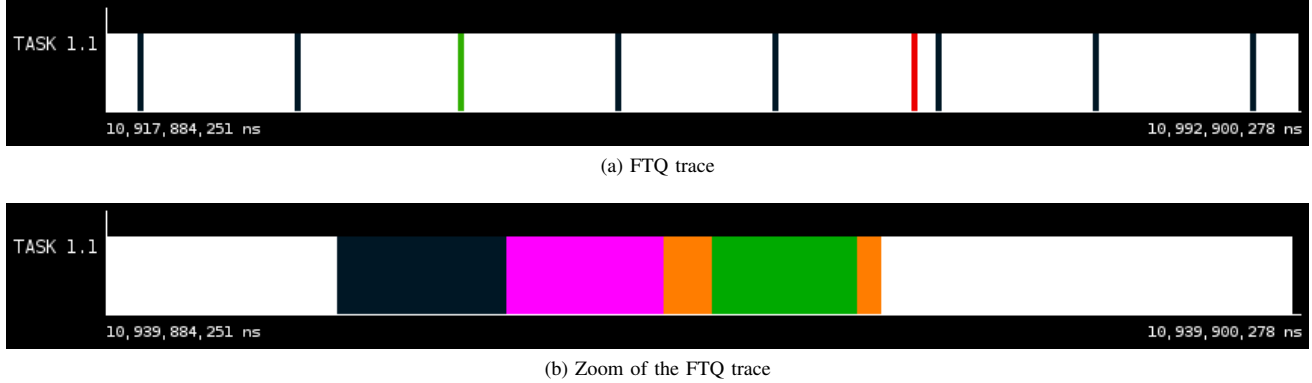


Fig. 2: FTQ Execution Trace. Figure 2a shows a part (75 ms) of the FTQ trace that highlights the periodic timer interrupts (black lines), the page faults (red line), and a process preemption (green line). Figure 2b zooms in and shows that the interruption consists of several kernel events. At this level of detail we can distinguish the timer interrupt (2.178  $\mu\text{sec}$ , black) followed by the `run_timer_softirq` softirq (1.842  $\mu\text{sec}$ , pink), the first part of the schedule (0.382  $\mu\text{sec}$ , orange), the process preemption (2.215  $\mu\text{sec}$ , green), and the second part of the schedule (0.179  $\mu\text{sec}$ , orange).

all those activities that are not explicitly requested by the applications but that are necessary for the correct functioning of the compute node. Timer interrupts, scheduling, page faults are examples of such activities. Second, we only account kernel activities as introduced noise when an application’s process is runnable. During the offline OS noise analysis, we do not consider a kernel interruption as noise if, when it occurs, a process is blocked waiting for communication.

Even with these modifications the number of kernel activities tracked is considerable. Since it is not possible to show all the tracked activities, in this paper we focus on those that appear more often or have large impact (such as timer interrupts, scheduling, and page faults). However, developers concerned about specific areas can use our infrastructure to drill down into any particular area of interest by simply applying different filters.<sup>2</sup>

The second extension we made to LTTng is an offline trace transformation tool. We developed an external LTTng module that generates execution traces suitable for Paraver [20]. Additionally, the module generate a data format that can be used as input to Matlab. We use this to derive the synthetic OS noise chart and the other graphs presented in this paper. We took particular care of nested events, i.e., events that happen while the OS is already performing other activities. For example, the local timer may raise an interrupt while the kernel is performing a tasklet. Handling nested events is particularly important for obtaining correct statistics.

### B. Tracing scalability

The amount of data generated to trace OS events on a single node is not an issue. On the other hand, the application of any trace methodology to clusters composed by a large number of nodes (i.e. thousands of nodes), face the challenge of collecting and storing a very large amount of data at run-time. This problem arises for any approach willing to capture OS noise events with a fine granularity in HPC systems.

<sup>2</sup>Filters are common features for HPC performance analysis tools and we provide the same capability for our Matlab module.

Given that OS noise is inherently redundant across nodes, one of the most effective solution is to enable tracing only on a statistically significant subset of the cluster’s nodes. Another option is to apply data-compression techniques at run-time to reduce the data-size.

### C. Analyzing FTQ with LTTNG-NOISE

A comparison of Figure 1a and Figure 1b shows that LTTNG-NOISE captures the OS noise identified by FTQ. The small differences between the two graphs can be explained by realizing that FTQ computes missing integral number of basic operations and multiplying this by the cost of each operation thus producing discretized values, while LTTNG-NOISE calculates the measured OS noise between given points using the trace events. In general, the result is that FTQ slightly overestimates the OS noise, for FTQ does not account for partially completed basic operations.

Apart from these small differences, the figures show that the data output from these two methods are very similar. Thus, we have a high degree of confidence in comparing results from the different techniques, and therefore that our new technique represents an accurate view of induced OS noise. Unlike FTQ, our new technique allows quantifying and providing details for what contributed to the noise of each interruption.

The Synthetic OS noise chart in Figure 1b shows, for each OS interruption, the kernel activities performed and their durations. For example, at time  $x = X_1$  FTQ detects 7.70  $\mu\text{sec}$  OS overhead (this point is showed in Figure 1a). The corresponding point in Figure 1b (also highlighted on the chart) shows an overhead of 6.96  $\mu\text{sec}$  but also shows that the interruption consists of `timer_interrupt` handler (1), a `run_timer_softirq` softirq (2), and a process preemption (`eventd` daemon) (3). Figures 1c and 1d show details of Figures 1a and 1b, respectively, centered around point  $x = X_1$ . Figure 1d shows that, indeed, the small frequent spikes are related to the timer interrupt handler but, also, to the `run_timer_softirq` softirq, which takes about the same amount of time. Figure 1d also shows that there are smaller

spikes, to the best of our knowledge not identified in previous work, that are similar to those generated by the timer interrupt handler, but that are not related to timers or other periodic activity. These interruptions are caused by page faults, which are, as we will see in Section IV, application-dependent.

The synthetic OS noise chart is useful to analyze the OS noise experienced by one process. HPC applications, however, consist of processes and threads distributed across nodes. Execution traces can be used to provide a deeper understanding of the relationships among the different application’s processes. Moreover, performance analysis tools, such as Paraver, are able to quickly extract statistics, zoom on interesting portion of the application, remove or mask states, etc. Figures 2a and 2b show a portion of the execution trace of FTQ. The execution traces generated with LTTNG-NOISE are very dense, even for short applications. Even if performance analysis tools are able to extract statistics and provide 3D views, the visual representation, especially for long applications, is often complex and lossy due to the number of pixels on a screen and the amount of information to be displayed at each time interval. For this reason, in this paper, we usually zoom in to show small, interesting portions of execution traces that highlight specific information or behavior.

Figure 2a shows 75 msec of FTQ execution. In this trace white represents the application running in user mode, while the other colors represent different kernel activities. The trace shows the periodic timer interrupts (black lines) and the frequent page fault (red lines). While, in this case, page faults take approximately the same time, timer interrupts may trigger other activities (e.g., softirqs, scheduling, or process preemption) and, thus, have different sizes. In order to better visualize the activities that were performed on a giving kernel interruption, Figure 2b shows a detail of the previous picture. In particular, Figure 2b depicts one interruption caused by a timer interrupt. Paraver provides several informations for each event, such as time duration or internal status, simply by clicking on the desired event. Regarding the detailed execution trace of FTQ, the tool provides the following time durations for each interruption: 2.178  $\mu$ sec (timer interrupt), 1.842  $\mu$ sec (run\_timer\_softirq), 0.382  $\mu$ sec (first part of the schedule that leads to a process preemption), 2.215  $\mu$ sec (process preemption), and 0.179  $\mu$ sec (second part of the schedule that resumes the FTQ process).

#### IV. EXPERIMENTAL RESULTS

In this section we present our experimental results based on LTTNG-NOISE and the Sequoia benchmarks [13] (*AMG*, *IRS*, *LAMMPS*, *SPHOT*, *UMT*). We ran experiments on a dual Quad-Core AMD Opteron(tm) processor workstation (for a total of 8 cores) with 64 Gigabyte of RAM. We used a standard Linux Kernel version 2.6.33.4 patched with LTTNG-NOISE. In order to minimize noise activity and to perform representative experiments, we removed from the test machine all kernel and user daemons that do not usually run on a HPC compute node. The system is in a private network, isolated from the outside world, that consists of the machine itself

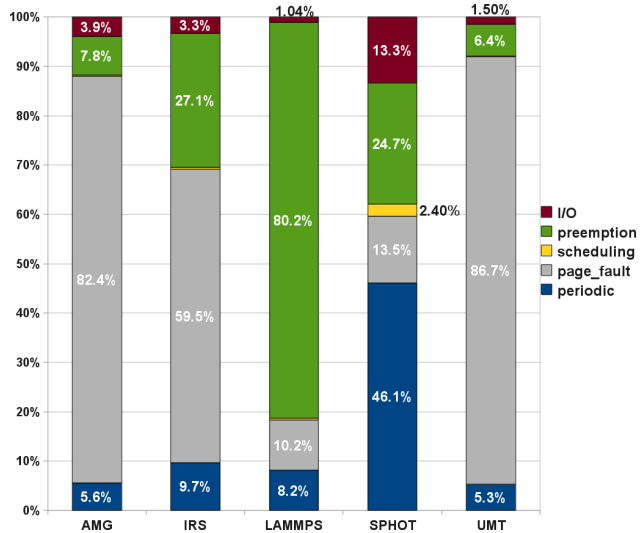


Fig. 3: OS noise breakdown for Sequoia benchmarks

and an NFS server. Most of the I/O operations, including the application’s executable files and the input sets, are performed through the network interface, as it is often the case for HPC compute nodes. In order to obtain representative results, Sequoia benchmarks were configured to run with 8 MPI tasks (one task per core), and to last several minutes.

Section IV-A analyzes the OS noise breakdown for each application. Sections IV-B, IV-C, IV-D, and IV-E analyze in detail the OS activities identified by our infrastructure as the major source of OS noise. For lack of space, we cannot put all the graphs and data we collected. In each sub-section we provide synthetic statistical data and a few examples of more detailed information for chosen interesting applications.

##### A. Noise breakdown

Each OS interruption may consists of several different kernel activities. Our analysis allows us to break down each OS interruption into kernel activities. For simplicity and clarity, we focus on the kernel activities with larger contribution to OS noise. To this extend, we classified kernel activities into 5 categories:

**periodic:** timer interrupt handler and run\_timer\_softirq, the softirq responsible to execute expired software timers.

**page fault:** page fault exception handler.

**scheduling:** the schedule function and the related softirqs (rcu\_process\_callbacks and run\_rebalance\_domains).

**preemption:** kernel and user daemons that preempt the application’s processes.

**I/O:** network interrupt handler, softirqs and tasklets.

Figure 3 shows the OS noise breakdown for the Sequoia applications. Each Sequoia application experiences OS noise in a different way. For example, page faults represent a

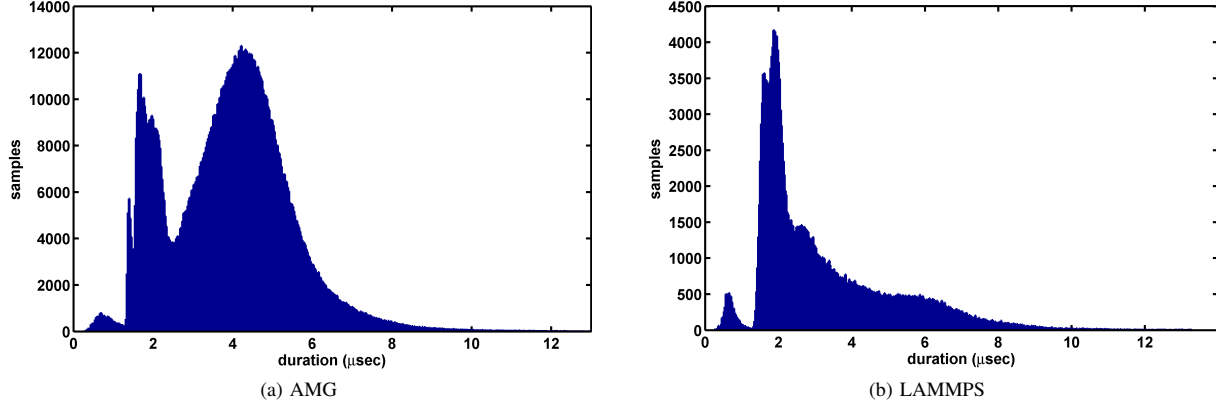


Fig. 4: Page fault time distributions

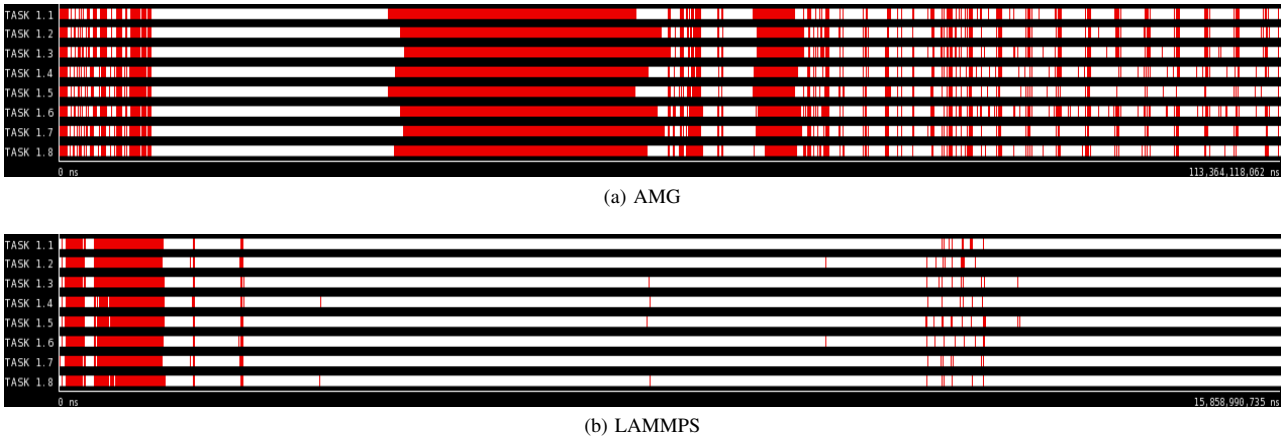


Fig. 5: Page Fault Trace. Figures 5a and 5b show the execution trace of *AMG* and *LAMMPS*, respectively. We filtered out all the events but the page faults (red). The traces highlight the different distributions of page fault for *AMG* (throughout all the execution) and *LAMMPS* (mainly located at the beginning and the end). Notice that in some regions page faults are very dense and appear as one large page fault when, in fact, there are thousands very close to each other but the pixel resolution does not allow distinguishing them.

large portion of OS noise for *AMG* and *UMT* (82.4% and 86.7%, respectively), while *SPHOT* and *LAMMPS* are only marginally affected by the same kernel activity (13.5% and 10.2%). *IRS*, *SPHOT* and, especially, *LAMMPS* are preempted by other processes during their computation, which represent a considerable component of their total jitter (27.1%, 24.7%, and 80.2%, respectively). Our analysis shows that the applications were interrupted particularly by *rpciod*, a I/O kernel daemon. Periodic activities (timer interrupt handler and periodic software timers) are, instead, limited (between 5% and 10%) for all applications but *SPHOT*.

### B. Page faults

Page faults can be one of the largest source of OS noise. Memory management functionalities like page-on-demand and copy on write can significantly reduce the impact of page faults. Moreover, one of the main advantages of lightweight kernels is to drastically simplify dynamic memory management, thus reducing or even removing (as in CNK [6]) the noise caused by page faults.

As Table I shows, for some applications, like *AMG*, *IRS*, and *UMT*, the frequency of page faults is even higher than

TABLE I: Page fault statistics

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	1,693	4,380	69,398,061	250
IRS	1,488	4,202	4,825,103	218
LAMMPS	231	3,221	27,544	248
SPHOT	25	2,467	889,333	221
UMT	3,554	4,545	50,208	229

that of the timer interrupt (10kHz in our configuration). Even more important is that the time distribution is very large and differs from application to application. From Table I, we can observe that each application presents a different number of page faults. Moreover, though the minimum duration of a fault is similar across the benchmarks (about 250 *nsec*), the maximum duration varies from application to application (from 25.7  $\mu$ sec to 69,398  $\mu$ sec). OS noise activities that vary so much may limit application scalability on large machines, as show in previous work [1], [4].

Figures 4a and 4b<sup>3</sup> show the page fault time distributions for

<sup>3</sup>Time distributions may have a very long tail that could make visualization difficult. To improve the visualization, we cut all the distributions in the histograms at the 99<sup>o</sup> percentile.

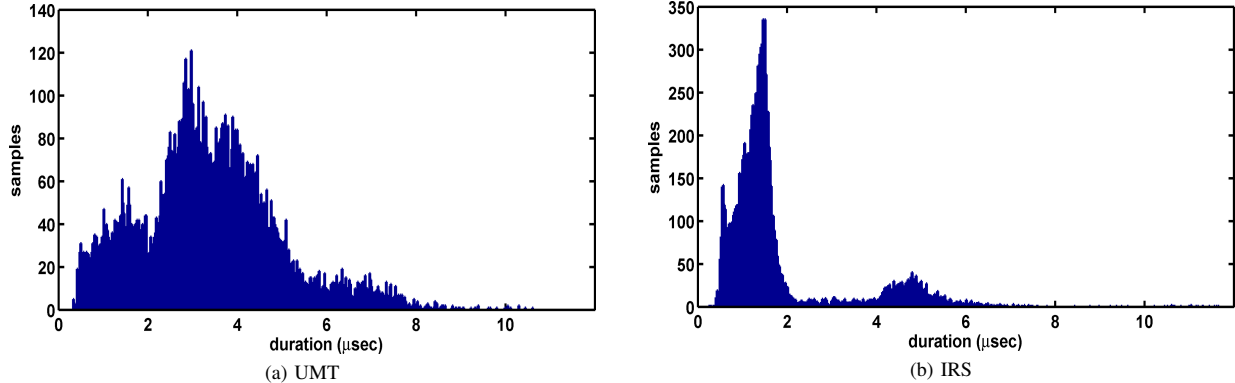


Fig. 6: Domain rebalance softirq time distribution

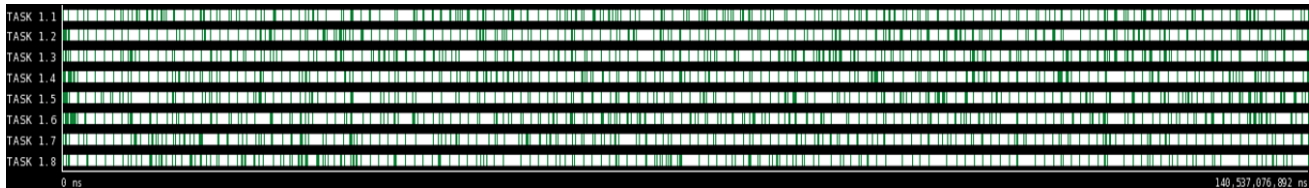


Fig. 7: Process preemption experienced by *LAMMPS*. This picture shows the complete execution trace of *LAMMPS*. We filtered out all events but process preemptions (green). Though the pixel resolution does not allow distinguishing all the process preemption, it is clear that *LAMMPS* suffers many frequent preemptions.

*AMG* and *LAMMPS*, respectively. We chose these two applications because they present different distributions. Figure 4a (*AMG*) shows two main picks (around  $2.5 \mu\text{sec}$  and  $4.5 \mu\text{sec}$ ), with a long tail. *LAMMPS*, on the other hand, shows a one-sided distribution with a main pick around  $2.5 \mu\text{sec}$ .

Figure 5 shows another important difference between *AMG* and *LAMMPS*. While for *LAMMPS* page faults, marked in red in the pictures, are mainly located at the beginning (initialization phase), *AMG* page faults are spread throughout the whole execution, with several accumulation points. From these execution traces we can conclude that page faults are not of main concerns for *LAMMPS*, neither in terms of OS noise overhead (10.2% of the total OS noise, as reported in Figure 3) nor from the time distribution, as they mainly appear during the initialization phase. For *AMG*, instead, page faults may seriously affect performance. They represent a considerable portion of the total OS noise (82.4%), and may interrupt application’s computing phases.

### C. Scheduling

Scheduling activities are not only related to the `schedule` function but also to other kernel daemons and softirqs that are responsible, among other tasks, to keep the system balanced.

Our analysis shows that, indeed, the overhead introduced by the `schedule` function is negligible and constant, confirming the effectiveness of the the new Completely Fair Schedule (CFS), which has a  $O(1)$  complexity.

Domain balancing, instead, may create several problems in terms of execution time variability and, therefore, scalability [24]. In this section we analyze

the effect of the `run_rebalance_domains` softirq. `run_rebalance_domains` is triggered when needed from the scheduler tick and it is in charge of checking whether the system is balanced and, if not, move tasks from one CPU to another. Domain rebalance is described in [24], [39], [42]. Here it is worthwhile to notice that rebalancing domains introduces two kinds of overheads: *direct* (time required to execute the rebalance code) and *indirect* (moving a process to another CPU may require extra time to warm up the local cache and other processor resources).

Figures 6a and 6b show the execution time distribution of the `run_rebalance_domains` softirq for *UMT* and *IRS*. While *IRS* shows a fairly compact distribution with a main pick around  $1.80 \mu\text{sec}$ , *UMT* shows a much larger distribution with average of  $3.36 \mu\text{sec}$ . Indeed, *UMT* is a complex application that involves, besides MPI, Python and pyMPI scripts. The OS has, thus, a much tougher job to balance *UMT* than *IRS*.

TABLE II: Network interrupt events frequency and duration

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	116	1,552	347,902	540
IRS	87	1,666	353,294	521
LAMMPS	11	2,520	356,380	594
SPHOT	21	1,372	341,003	535
UMT	77	1,975	349,288	484

### D. Process preemption and I/O

The OS scheduler may decide to suspend one or more running processes during the execution of a parallel application

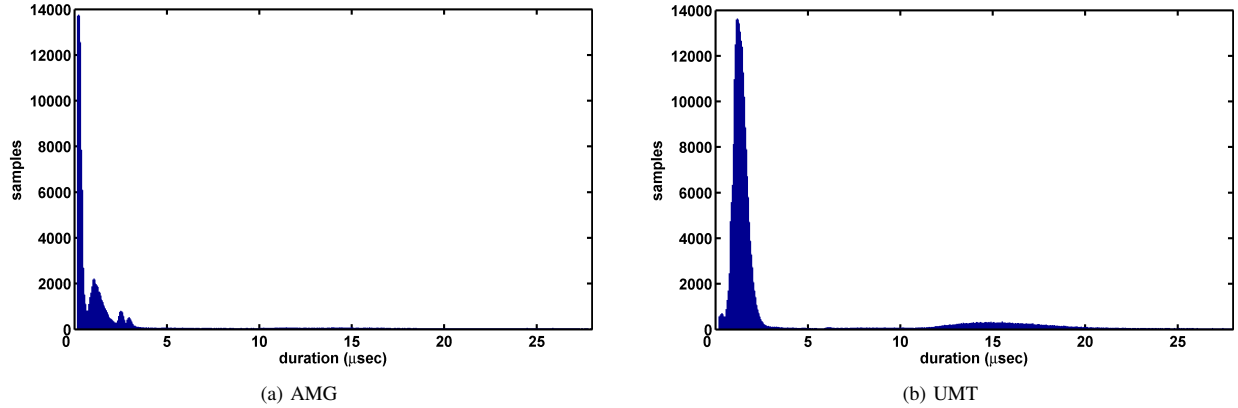


Fig. 8: run\_timer\_softirq time distribution

TABLE III: net\_rx\_action frequency and duration

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	53	3,031	98,570	192
IRS	43	4,460	78,236	174
LAMMPS	10	4,707	84,152	199
SPHOT	15	1,987	45,150	207
UMT	22	5,484	75,042	167

TABLE IV: net\_tx\_action frequency and duration

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	15	471	8,227	176
IRS	10	504	4,725	176
LAMMPS	2	559	4,392	175
SPHOT	3	409	2,746	200
UMT	9	545	8,902	173

(*process preemption*). The suspended process is not able to progress during that time and the whole parallel application may be delayed. Typically, the OS suspends a process because there is another higher-priority process that needs to be executed. Kernel and user daemons are classical examples of such processes. The interrupted process may, in turn, be migrated by the scheduler on another CPU. This approach provides advantages if the target CPU is idle but may also force time sharing with another process of the parallel application. A discussion of the possible effect of process migration and preemption is provided in [24].

In our experiments the only kernel daemon that is very active is the I/O daemon (`rpciod`).<sup>4</sup> HPC compute nodes do not usually have disks or other I/O devices except, of course, the network adapter. Most HPC networks (such as Infiniband [43] or BlueGene Torus [44]) allow user applications to send/receive messages without involving the OS, through kernel bypass. All I/O operations (e.g., reading input data or

<sup>4</sup>LTTNG-NOISE also uses a kernel daemon to collect data at run time. This daemon is not supposed to be running in a normal environment, therefore we are not taking it into account in the rest of the discussion. However, we noticed that, though it depends on the application, the LTTNG-NOISE kernel daemon was especially active at the beginning and at the end of the applications. Moreover, as reported in section III, the overall overhead of LTTNG-NOISE is quite small.

writing final results) are shipped to an I/O node through the network. Once the operation is completed, the I/O node returns the results to the compute node.

In our test environment, the compute node is connected to an NFS server through the `rpciod` I/O daemon. For most of the applications, `rpciod` is the only kernel daemon that generate OS noise. *UMT* is a different case because the application is more complex than the others. In particular, *UMT* runs several Python processes that may 1) interrupt the computing tasks, and 2) trigger process migration and domain balancing.

Figure 3 shows that the OS noise experienced by *LAMMPS* is dominated by process preemption, marked in green. Figure 7 shows that, indeed, *LAMMPS* processes are preempted several times throughout the execution. As opposed to the other benchmarks, *LAMMPS* performs a considerable amount of I/O operations. Since data are moved to/from the network, the OS suspends *LAMMPS* processes that issued I/O operations until the data transfer is completed. In Linux, network I/O operations involve the network interrupt handler and the receiver (`net_rx_action`) and transmission (`net_tx_action`) tasklets.<sup>5</sup> On data transfer completion, the active network tasklet wakes up the suspended processes in the order I/O operations complete and on the CPU that receives the network interrupt. Clearly, that CPU may be running another *LAMMPS* process (which is preempted) at that time, thus a load balance reschedule may be triggered to move the preempted process on an idle CPU (migration).

Tables II, III and IV report the frequency and time duration of the network interrupt handler, the `net_rx_action`, and the `net_tx_action` tasklet, respectively. As we can see, the transmission tasklet is faster and more constant than the receiver tasklet. The reason is that while sending data to the NFS server is an asynchronous operation, receiving data from the NFS server must be done synchronously. In fact, the transmission tasklet can return right after the network DMA engine has been started, because the copy of the data from

<sup>5</sup>Tasklets are implemented on top of softirqs and differ from the latter in that tasklets of the same type are always serialized. In other words, two tasklets of the same type cannot run on two different CPUs while two softirqs of the same type are allowed to do so [39].

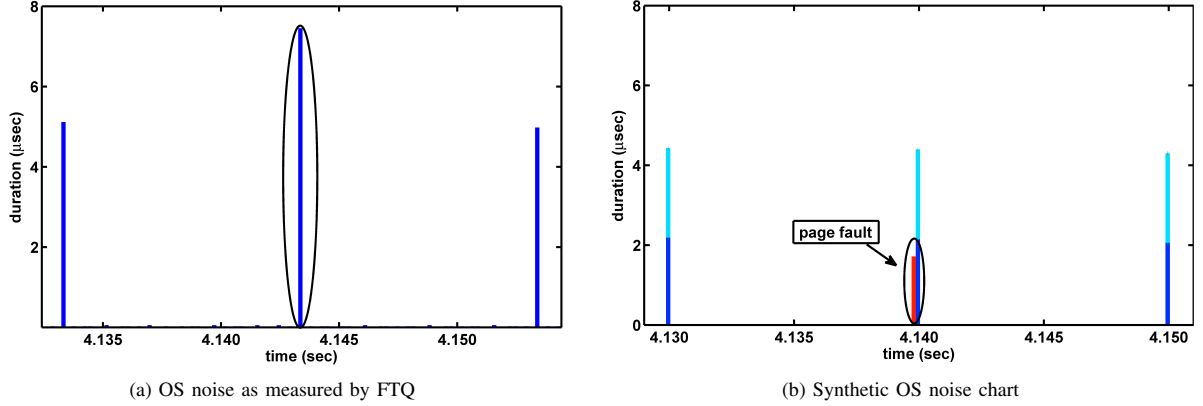


Fig. 9: Noise disambiguation

memory to the network adapter’s buffer will be performed asynchronously by the DMA engine. When receiving a message, instead, the receiving tasklet needs to wait until the data transfer from the network buffer to memory has completed or else the application may find corrupted data.

#### E. Periodic activities

Some activities are periodically started by the timer interrupt. With the introduction of high resolution timers in Linux 2.6.18, the local timer may raise an interrupt any time a high resolution timer expires. One of such timer is the *periodic timer interrupt*, which is in charge of accounting for the current process elapsed time and, eventually, call the scheduler if the process time quantum has expired. This interrupt is a well-known source of OS noise [4], [5]. In our test machine we set the frequency of this periodic high resolution timer to the lowest possible (10 kHz) so to minimize the effect of the periodic timer interrupt.

The term *timer interrupt* is often used to identify both the timer interrupt and the softirq. With our methodology, instead, we are able to distinguish between the timer interrupt (or *timer interrupt top half*) and the softirq `run_timer_softirq` (called *bottom half* in old Linux releases). The `run_timer_softirq`, in particular, may vary considerably across applications and between two separate instances of the event. This softirq, in fact, is in charge of running all the handlers connected to expired software timers. Each handler may have a different duration and applications may set software timers accordingly to their needs (for example, to take regular check points). It follows, that this activity may show a considerable execution time variation.

TABLE V: Timer interrupt statistics

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	100	3,334	29,422	795
IRS	100	6,289	35,734	867
LAMMPS	100	3,763	34,555	1,194
SPHOT	100	1,498	10,204	833
UMT	100	6,451	29,662	982

TABLE VI: Softirq `run_timer_softirq` statistics

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	100	1,718	49,030	191
IRS	100	3,897	57,663	193
LAMMPS	100	2,242	58,628	256
SPHOT	100	620	32,926	223
UMT	100	3364	87,472	214

Figures 8a and 8b show the execution time distribution of the `run_timer_softirq` softirq for *AMG* and *UMT*. As we can see from the figures, and as confirmed from previous studies, the `run_timer_softirq` softirq has a long-tail density function.

Table V and VI show statistical data related to the timer interrupt and `run_timer_softirq` softirq. As expected, the frequency of the timer interrupt is at least 100 events/second (10 kHz) for all the applications. Also, the fact that the frequency is not higher means that the applications do not set any other software timer.

## V. NOISE DISAMBIGUATION

Analyzing OS noise indirectly through micro benchmarks may hide the true causes of a given noise event and lead the developer towards the wrong direction. This section provides two example of using LTTNG-NOISE to disambiguate similar OS activities.

#### A. Disambiguation of qualitative similar activities

Figure 10 shows a portion of the synthetic OS noise chart for *AMG*. The graphs shows several page faults (red), and two timer interrupts (blue) which, in turn, trigger the `run_timer_softirq` softirq (green). The picture shows that there are two interruptions, a page fault and a timer interrupt (both highlighted in the graph) that have similar duration (2913 nsec and 2902 nsec, respectively). Indirect measurements through micro benchmarks do not permit to distinguish between these two kernel activities, thus, the developer may think that they are the same activity.

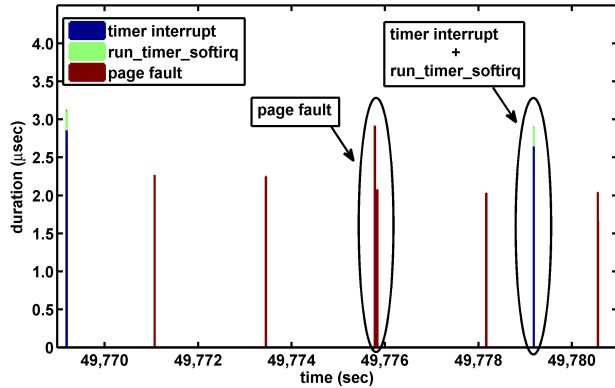


Fig. 10: AMG - Synthetic OS noise graph

The synthetic OS noise chart provided by LTTNG-NOISE, instead, allows us to disambiguate the two OS interruptions and to clearly identify each activity. In this particular case, the graph shows that the first highlighted interruption is a page fault that takes 2913 *nsec*, while the second interruption is composed by a timer interrupt handler (2648 *nsec*) and `run_timer_softirq` softirq (254 *nsec*), which sum up to a total of 2902 *nsec*.

### B. OS noise composition

Micro benchmarks, such as FTQ, compute the OS noise as missing operations in a given iteration [37]. This means that micro benchmarks are not able to distinguish two unrelated events if they happen in the same iteration.

Figure 9a shows three iterations of the FTQ micro benchmark. The three spikes are equidistant (which suggests a common periodic activity) but the jitter measured in the first and the third iterations (about 5  $\mu\text{sec}$ ) is different from the one measured in the second iteration (7.5  $\mu\text{sec}$ ). This different OS noise suggests that the events occurred during the first and the third iterations are different from those occurred during the second iteration. Moreover, the spikes in the first and third iterations are similar to the very frequent ones caused by the timer interrupt while the spike in the third iteration is not similar to anything else. A qualitatively analysis would conclude that FTQ experienced a timer interrupt during the first and the third iteration while “something else” happened during the second iteration, contradicting the hypothesis of equidistant events.

Our analysis shows that, indeed, a timer interrupt also occurred during the second iteration (Figure 9b), confirming the first observation of equidistant events. However, right before that timer interrupt, a page fault occurred. In this case, FTQ was not able to distinguish the two events that, indeed, appear as one in its graph. LTTNG-NOISE, instead, precisely shows the two events as separate interruptions, allowing the developer to derive correct conclusions about the nature of the OS noise. This example shows how an internal analysis is more effective than an indirect one and provides information that, otherwise, would be impossible to obtain.

## VI. CONCLUSIONS

OS noise has been studied extensively and it is a well-known problem in the HPC community. Though previous studies succeeded to provide important insights on OS noise, most of those studies are “qualitative” in that they characterize the overall effect of OS noise on parallel applications but tend not to identify and characterize each kernel noise event. As HPC applications evolve towards more complex programming paradigms that require richer support from the OS, we believe that a quantitative analysis of the kernel noise introduced by such operating systems is most needed.

In this paper we presented our technique to provide a quantitative descriptive analysis for each kernel noise event. We extended LTTng, a low-overhead, high-resolution kernel tracer, with extra trace points and offline modules to analyze the OS noise. In particular, we developed a module that generates execution traces suitable for Paraver and a data format that can be used as input for Matlab. We demonstrated that our new technique and toolkit (LTTNG-NOISE) accurately reflect the noise measured by other known techniques (such as FTQ). In addition to previous insights, our new technique allows quantifying and providing details for what contributed to the noise for each interruption.

Using LTTNG-NOISE, we analyzed the OS noise introduced by the OS on LLNL Sequoia applications and showed that 1) each application experiences different jitter (both in terms of overhead and composition), 2) page faults may have even larger impact than timer interrupts (both in terms of frequency and duration), and 3) some activities have larger time distributions that may lead to load imbalance at scale for particular applications. Moreover, we are able to identify and quantify each single source of OS noise. This capability is very useful to analyze current and future applications and systems.

Finally, we showed two case studies where we used our technique to disambiguate kernel noise events. This noise disambiguation would not be possible without the detailed information provided by LTTNG-NOISE.

We plan to use LTTNG-NOISE to do deeper analysis of current and future parallel applications and to quantify how our findings affect the scalability of those applications on large machines with hundreds of thousands of cores.

## ACKNOWLEDGMENTS

This work was supported by a Collaboration Agreement between IBM and BSC with funds from IBM Research and IBM Deep Computing organizations. It has also been supported by the Ministry of Science and Technology of Spain under contracts TIN-2007-60625 and JCI-2008-3688, as well as the HiPEAC Network of Excellence (IST-004408). The authors thank all the anonymous reviewers and the shepherd for constructive comments and suggestions.

## REFERENCES

- [1] 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 *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

- [2] K. B. Ferreira., P. G. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.
- [3] E. Shmueli, G. Almasi, J. Brunheroto, n. José Casta G. Doza, S. Kumar, and D. Lieber, "Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L," in *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 165–174.
- [4] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *The 4th IEEE Int. Symp. on Signal Processing and Information Technology (ISSPIT 2004)*, Rome, Italy, December 2004, <http://bravo.ce.uniroma2.it/home/gioiosa/pub/isspit04.pdf>.
- [5] D. Tsafir, Y. Etsion, D. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *ICS '05: Proc. of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: ACM Press, 2005, pp. 303–312.
- [6] M. Giampapa, T. Gooding, T. Inglett, and R. Wisniewski, "Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk," 2010.
- [7] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira, "Designing and implementing lightweight kernels for capability computing," *Concurr. Comput.: Pract. Exper.*, vol. 21, no. 6, pp. 793–817, 2009.
- [8] Argonne National Laboratory, Mathematics and Computer Science, "The Message Passing Interface (MPI) standard," <http://www-unix.mcs.anl.gov/mpl/>.
- [9] M. Forum, "MPI: A message passing interface standard," vol. 8, 1994.
- [10] NASA, "NAS parallel benchmarks," <http://www.nas.nasa.gov/Resources/Software/npb.html>.
- [11] L. Carrington, D. Komatitsch, M. Laurenzano, M. M. Tikir, D. Michea, N. Le Goff, A. Snively, and J. Tromp, "High-frequency simulations of global seismic wave propagation using SPECFEM3D\_GLOBE on 62k processors," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008.
- [12] E. Aprà, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas, "Liquid water: obtaining the right answer for the right reasons," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–7.
- [13] LLNL, "Sequoia benchmarks," <https://asc.llnl.gov/sequoia/benchmarks/>.
- [14] "NAMD scalable molecular dynamics," <http://www.ks.uiuc.edu/Research/namd/>.
- [15] "AMBER: Assisted model building with energy refinement," <http://ambermd.org/>.
- [16] D. J. Kerbyson, P. W. Jones, D. J. Kerbyson, and P. W. Jones, "A performance model of the parallel ocean program," *International Journal of High Performance Computing Applications*, vol. 19, 2005.
- [17] C. Bekas, A. Curioni, and I. Fedulova, "Low cost high performance uncertainty quantification," in *WHPCF '09: Proceedings of the 2nd Workshop on High Performance Computational Finance*. New York, NY, USA: ACM, 2009, pp. 1–8.
- [18] OpenMP Architecture Review Board, "The OpenMP specification for parallel programming," available at <http://www.openmp.org>.
- [19] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell, "Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing," *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.
- [20] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, S. Girona, and D. D. D. Computadors, "Paraver: A tool to visualize and analyze parallel code," In *WoTUG-18*, Tech. Rep., 1995.
- [21] A. Nataraj and M. Sottile, "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*, 2007.
- [22] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *Proc. of the 2007 IEEE Int. Conf. on Cluster Computing*, Austin, Texas, 2007.
- [23] T. Jones, S. Dawson, R. Neel, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10.
- [24] R. Gioiosa, S. McKee, and M. Valero, "Designing os for hpc applications: Scheduling," *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2010)*, September 2010.
- [25] P. Terry, A. Shan, and P. Huttunen, "Improving application performance on HPC systems with process synchronization," *Linux Journal*, November 2004, <http://www.linuxjournal.com/article/7690>.
- [26] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Proc. of the 2006 IEEE Int. Conf. on Cluster Computing*, Barcelona, Spain, 2006.
- [27] J. Liedtke, "Improving IPC by kernel design," in *SOSP '93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 1993, pp. 175–188.
- [28] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995, pp. 251–266.
- [29] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*. Orcas Island, Washington: IEEE Computer Society, May 1995, pp. 78–83.
- [30] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole, Jr., "The operating system kernel as a secure programmable machine," *Operating Systems Review*, vol. 29, no. 1, pp. 78–82, January 1995.
- [31] J. Appavoo, K. Hui, M. Stumm, R. W. Wisniewski, D. D. Silva, O. Krieger, and C. A. N. Soules, "An infrastructure for multiprocessor run-time adaptation," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM, 2002, pp. 3–8.
- [32] J. E. Moreira, M. Brutman, n. José Casta T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt, "Designing a highly-scalable operating system: the Blue Gene/L story," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 118.
- [33] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 1–12, 2006.
- [34] —, "Operating system issues for petascale systems," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 29–33, 2006.
- [35] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, 2008.
- [36] P. De, V. Mann, and U. Mittal, "Handling OS jitter on multicore multithreaded systems," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.
- [37] M. Sottile and R. Minnich, "Analysis of microbenchmarks for performance tuning of clusters," in *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 371–377.
- [38] E. Betti, M. Cesati, R. Gioiosa, and F. Piermaria, "A global operating system for HPC clusters," in *CLUSTER*, 2009, pp. 1–10.
- [39] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly, 2005.
- [40] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *Proceedings of the 2006 Linux Symposium*, 2006.
- [41] —, "LTTng, filling the gap between kernel instrumentation and a widely usable kernel tracer," in *Linux Foundation Collaboration Summit 2009 (LFCS 2009)*, Apr. 2009.
- [42] C. Boneti, R. Gioiosa, F. Cazorla, and M. Valero, "A dynamic scheduler for balancing HPC applications," in *SC '08: Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
- [43] T. Shanley, *InfiniBand Network Architecture*. Mindshare, Inc., 2002.
- [44] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu, "Early evaluation of ibm bluegene/p," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008.