Characterization of Virtualization-Induced Noise in High-Performance Computing and Real-Time Virtual Machines

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Francesco Aristei, 968347

Advisor:
Prof. Vittorio Zaccaria

Co-advisors:
Paolo Bonzini
Daniel Bristot de Oliveira

Academic year:
2022-2023

Abstract:
The use of virtualization technologies in High Performance Computing and Real-Time applications has become an accepted practice. Indeed, the majority of the HPC applications are moving to the infrastructures offered by the Cloud, thanks to the cost savings and flexibility they bring. Exceptions come primarily from security-sensitive applications such as those in the Defense field or applications with nanosecond-level latencies that can be found, for example, in high frequency trading. At the same time, the use of virtual machines have gained popularity in Real-Time systems for the ease offered in integrating and supporting mixed-criticality systems. Finally, with the advent of 5G technology, Network Function Virtualization is gaining traction, with low-latency network services deployed on top of virtual machines. The scientific community has demonstrated how these kinds of workload achieve comparable performance when run on top of a virtual machine and on bare-metal. However, the standard tools and benchmarks used to debug the systems where these kind of applications need to run, do not focus on the impact that virtualization has on the overall performance, and do not provide the causes of performance degradations that virtualization introduces, when encountered. This work highlights the role virtualization plays in the performance deficits experienced by low-latency applications and improves the debugging of the systems where these workloads are executed. It does so by extending the Osnoise tracer, an in-kernel tracer in the Linux operating system, that allows both the measurement and the tracing of the Operating System Noise, a well-known problem in the HPC domain, with impact on Real-Time applications as well. Osnoise provides both the measurements and the tracing of the sources of the noise, in an integrated manner, facilitating the analysis and debugging of the system. Starting from this contribution, the work extends the tool following a similar approach, by first allowing to measure the impact that the virtualization has on the overall Operating System Noise experienced by a workload, and then providing the tracing information, to reconstruct the events that lead to the virtualization noise measured. It finally shows how the extension is practically useful to speed-up the debugging of the system where a virtual machine is under execution.

Key-words: KVM, Virtual Machines, Operating System Noise, High-Performance Computing, Soft Real-Time Systems

1. Introduction

Currently, the scientific community has broadly accepted the use of virtualization technologies in latency-sensitive applications [3, 33]. The main use cases one can think of come from the High Performance Computing (HPC) and Real-Time worlds. Indeed, HPC applications have moved from dedicated (and expensive) infrastructures to the low-cost and flexible alternative represented by the Cloud, virtualizing the HPC infrastructure, such as the computing nodes. All the major Cloud Providers allow to access the infrastructures needed to perform large, complex HPC tasks such as data storage, networking solutions, specialized compute resources, security, and artificial intelligence applications. This shift has involved the majority of the industries dealing with High Performance Computing problems, from the Aerospace sector, in which an HPC architecture is needed in order to address problem such as lift and propulsion maximization, to the Energy sector, where it is used in problems such as efficiency improvement of electrical grids, or in the Life Science domain, where HPC is fundamental to simulate complex systems like molecules or proteins. Virtual machines have gained popularity in real-time applications as a flexible way to integrate and reuse heterogeneous software components with mixed criticality levels while obtaining isolation guarantees. Indeed, in industrial domains such as avionics, automotive, and the Industrial Internet of Things (IIoT), it is quite common to deal with the so-called mixed-criticality systems. These systems integrate functionalities of different safety and / or time-criticalness into a common platform to reduce the size, weight, power, and cost of hardware. Virtualization supports mixed-criticality compositions since it implicitly provides software support for partitioning and running tasks on heterogeneous OS (real-time and general-purpose). With the advent of 5G technology, another important use case, between HPC and Real-Time, is networking; the most notable applications are Software Defined Networking (SDN) [15] and Network Function Virtualization (NFV) [41]. The former is a technology that makes a network programmable by separating the system that is going to decide where traffic should be sent, i.e., the control plane, from the underlying system that pushes packets of data to a particular destinations, i.e., the data plane. The latter is a way to virtualize network services, such as routers, firewalls, and load balancers, which have traditionally been run on proprietary hardware. These services are packaged as virtual machines (VMs) on commodity hardware, allowing service providers to run their network on standard servers instead of proprietary ones.

The main concern raised about the adoption of virtual machines in these kind of applications has been regarding the delays that virtualization introduces. Indeed, most of the activities of VMs, of privileged type, need to pass through the intervention of an additional software layer: the hypervisor. The hypervisor, or virtual machine monitor (VMM), provides an environment for programs which is essentially identical with the original machine. The VMM must have complete control over the system resources assigned to a specific guest [38]. Therefore, many of the instructions a virtual machine tries to execute, which need to access hardware resources, require the intervention of the VMM, which handles the instruction on behalf of the guest to avoid that the virtual machine accesses resources not allocated to it. Instructions like memory accesses, interrupt handling, or I/O operations need to pass through the intervention of the hypervisor, exiting the guest. This mechanism introduces non-negligible overhead.

However, the literature has extensively studied the feasibility of the adoption of virtual machines to perform low-latency tasks, with the conclusion that such delays are indeed acceptable [27, 30]. In the real-time domain, different metrics have been used to measure the performance differences between running a tool on a virtual machine and on bare metal. The scheduling latency [14] represents the time that the system is unproductive due to scheduling tasks, such as the time needed to make context switching and scheduling decisions. It has been measured on different hypervisors, showing that with the proper system configuration, the values obtained are close to the case of bare metal [7]. Another important latency metric coming from the HPC world, but also used in real-time applications is the Operating System Noise [13]. The OS noise that affects a workload is defined as the CPU time that the system steals from this workload while it is ready to run. Also in this case, operating system noise has been measured on top of a virtual machine, relying on different hypervisor technologies, with the conclusion that the values obtained are close to the case of a bare metal solution [34]. Moving to the HPC case, various benchmarks used to assess the performance of parallel computing systems have been run in virtual environments and in bare metal with comparable results [17].

However, in both communities, the studies have been conducted mainly using a black-box approach. Specifically, the standard tools used to conduct these types of study only provide the values of the metrics they are measuring. Therefore, performance deficits are highlighted, in terms of difference between guest and bare metal case, without explaining what caused them. Moreover, none of these tools provides additional information when run on top of a virtual machine, to account for the impact that the hypervisor and, more generally, the virtualization have on the performance. The user is left with the task of analyzing the execution environment to understand the origins of such performance degradations. In both the HPC and the real-time domains, the Linux Operating System (OS) [9] has proven to be the best viable option for a wide range of very niche applications, despite its general-purpose nature. For example, Linux can be found in the High-Performance Computing (HPC) domain, running on all the top 500 supercomputers [44]. It can also be found in the domain of embedded real-time systems, not only in the area of industrial automation and robot control, but even in space [45]. These achievements are possible due to the great flexibility in the configuration options of Linux and specifically its kernel. Osnoise [13] is a tool integrated into the tracing facilities of the Linux kernel that allows the measurement of operating system noise. The tool is able to provide a quantitative view of that metric, together with the tracing information, linking in an unambiguous way the delays measured with the events that caused them. In fact, it comes with a trace-based and workload-based component. Workload-based components simulate a workload capable of accounting for the OS noise measurement as a metric reported by the workload. Specifically, it detects a large amount of time elapsed between two consecutive reads of the time. Instead, the trace-based component uses the Linux tracing infrastructure to show potential causes of latency spikes. Osnoise integrates the two, leveraging both the workload and a tracing mechanism synchronized together to account for operating system noise while still providing detailed information on the root causes of OS noise spikes.

As already explained, in both HPC and real-time systems deployed within virtualized environment, performance metrics, including operating system noise, are measured considering the system as a black box, thus without providing insights on the events affecting the performance. Operating system (OS) noise (or jitter) is a well-known problem in the High Performance Computing (HPC) community. Petrini et al. [22] showed how OS noise can limit the application's scalability, severely reducing performance on large-scale machines and large-system activities (e.g., network file system server); therefore, every HPC workload running in a virtualized environment would benefit from the analysis provided by osnoise, which, differently from the standard tools used in the industry, links the detected noise with its causes.

Moving on the boundaries of HPC and real-time systems, Network Function Virtualization is the process of replacing network appliance hardware with virtual machines. Virtual machines use a hypervisor to run networking software and processes such as routing and load balancing. These workloads are expected to run with tight timing requirements. To achieve high throughput, the generic network stack of the operating system is often bypassed, and all network packet processing is done in the user space by a specific process that handles the network flows. These kinds of workload are highly sensitive to interference caused by the hypervisor and other higher-priority tasks. Having a tool that can highlight this noise introduced by the virtualization layer allows virtual machines to be tuned and speeds up the debugging of the system, reaching real-time performance. However, as of today, when run on top of a virtual machine, osnoise is not able to provide insightful information regarding the noise caused by the virtualization layer, such as the time stolen from the guest, executing the hypervisor, or other higher-priority tasks. Indeed, as with the other tools, it does not add any information to account for the impact of virtualization on the noise experienced by a workload. Having this kind of knowledge would suggest to the user where the noise has originated, improving the debugging of the system. In fact, this source of disturbance is classified as generic hardware noise by the tool, giving a misleading interpretation of what causes system delays. Therefore, this work proposes to start from osnoise, as a tool able to give a complete understanding of the noise affecting an executing workload; thanks to its hybrid approach that mix together a workload-based and tracing-based component giving both a quantitative and qualitative view of the noise. From there, the work extends the tool in order to account for the noise that working in a virtualized environment introduces, adding information about the role of the hypervisor and the host in determining the overall noise. Before describing more in detail the work performed, it follows a brief summary of each of the contribution made.

- The tool has been extended both on the guest side and on the host side. In the former case to provide a quantitative view of the virtualization noise, in the latter, to better understand the causes of the noise measured on the guest.
- Several experiments have been performed to verify that the results provided by the introduced extensions were consistent with the expected behavior.
- A validation step concludes the work, where experiments have been made in order to highlight the practical utility of what has been done. The tool has been used to measure the virtualization noise affecting a virtual machine, and then to understand the events that are responsible for that noise, showing a practical usage in systems debugging.

Being Linux the operating system of choice, the virtualization technology on which the extension is based is the Kernel Virtual Machine (KVM) [4], a module of the Linux system that turns it into a hypervisor. KVM allows Linux to use hardware-assisted virtualization technology to create and manage virtual machines. For I/O emulations, KVM uses a userland software, QEMU [8], which performs, as said, hardware emulation. The targeted architecture on which the work has been developed is the x86_64 with vTX as hardware-assisted virtualization technology [21].

As a first step in the extension process, when running inside the guest, new statistics have been added that measure the noise caused by the hypervisor and the host. This gives the user a quantitative view of virtualization noise. When osnoise is launched on the host, while one or more virtual machines are executing, a new tracepoint keeps track of each time the vCPUs are kicked off the physical processor in order to execute other tasks. This, combined with the tracing information provided by osnoise, allows one to identify the events that caused the noise measured on the guest. In this way, the user has both a measure of virtualization noise and the root

causes of it.
A series of experiments has been performed. First, we verify that the output provided by osnoise is consistent with the expected results. The hardware noise, caused by unmaskable hardware interrupts, such as System Management Interrupts (SMIs), is the same in the host and the guests running on top of it because it is related to the hardware and it does not depend on the virtualization process. Therefore, osnoise has been run on top of a virtual machine in both the vanilla version and the new version. As expected, the measured hardware-induced noise has dropped significantly from the first to the second case, stating that much of the hardware noise measured in the vanilla version was due to the virtualization layer. This shows that the extension introduced allows one to better distinguish between the hardware noise and the noise introduced by virtualization. More importantly, the hardware noise measured in the last case is equivalent to the one obtained running osnoise directly on the host, ensuring that the results are coherent with the expected behavior.
Finally, it has been shown how osnoise can be practically used to debug and tune a system. To run low-latency applications, both hardware and Linux have been configured according to standard practices in both the HPC and the real-time domains. To this end, the hardware is configured to achieve the best trade-off between performance and determinism. This setup includes adjusting the processor speed and power savings setup while disabling features that could cause hardware-induced latencies, such as system management interrupts (SMIs). Regarding the Linux configuration, the system is usually partitioned into a set of isolated and housekeeping CPUs, which is a typical setup for HPC systems. Housekeeping CPUs are those where the tasks necessary for the regular system usage will run. This includes kernel threads responsible for in-kernel mechanisms, such as RCU (read-copyupdate) callback threads [36], kernel threads that perform deferred work, such as kworkers, and threads dispatched by daemons and users. The general system IRQs (Interrupt ReQuests) are also routed to housekeeping CPUs. In this way, the isolated CPUs are then dedicated to low-latency tasks. However, despite the high-grade CPU isolation level currently available on Linux, some housekeeping work is still necessary on all CPUs. For example, the timer IRQ still needs to happen under certain conditions, and some kernel activities need to dispatch a kworker for all online CPUs. Finally, drawing on real-time setups, low-latency workloads such as those in the NFV world are often configured with real-time priorities, and the kernel is generally configured with fully preemptive mode (using the PREEMPT RT patchset [23]) to provide bounded wake-up latencies. At this point, osnoise was run with various tuning configurations in order to understand their impact and the feasibility of running low-latency tasks on virtualized platforms. Initially, CPU isolation was applied solely to the host system, followed by isolation only in the guest, and lastly, the most rigorous isolation scenario was tested by tuning both the host and guest systems.
Even if it has considerably improved with respect to the non-tuned case, the resulting noise obtained after tuning both the host and the guest exceeds the values requested to perform latency-sensitive workloads. Therefore, a deeper analysis was performed. Specifically, a virtual machine was launched with CPU isolation applied to both the host and the virtual instance. Osnoise was run on isolated vCPUs within the guest, while osnoise was run simultaneously on the host. By looking at the tracing information given by the host extensions of osnoise, it is possible to deduce that the cause is the higher priority tasks preempting the vCPUs where the guest is executing. This is due to the hardware used to perform the experiments. In fact, with only four available cores, the system is overloaded, forcing the operating system to execute housekeeping jobs on isolated CPUs. This shows how osnoise facilitates debugging the system, guiding the user where the noise originates. The experimental results confirm the validity of the output provided by osnoise, in the sense that it is coherent with the expected behavior, and show how it can speed up the debugging and tuning of the system, providing both a quantitative and qualitative view of the noise.

2. Background

In order to understand the work performed, is first necessary to briefly describe the tool on which the extension has been performed: osnoise. After that, an introduction is followed on how virtual machines work, with a focus on the noise introduced by the virtualization process and on the virtualization technology of the Linux operating system: the Kernel Virtual Machine (KVM).

2.1. osnoise

OS noise, sometimes called OS jitter, is a well-known problem in the HPC field [5, 22]. Generally, HPC workloads follow the single-program multiple-data (SPMD) model, shown in Figure 1.

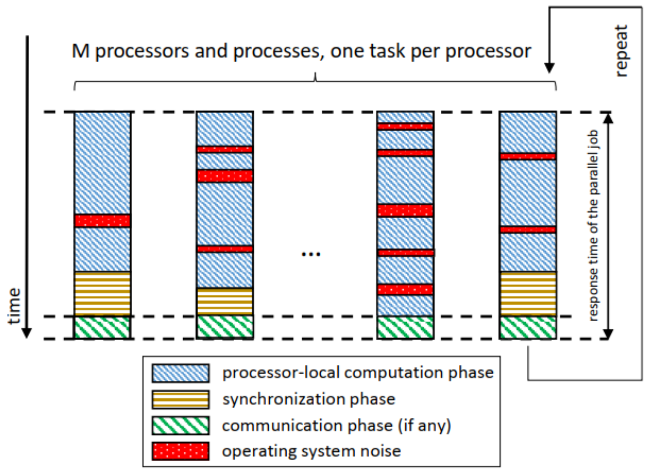

M processors and processes, one task per processor
repeat
time
...
response time of the parallel job
processor-local computation phase
synchronization phase
communication phase (if any)
operating system noise

Figure 1: The single-program multiple-data (SPMD) model used in HPC workloads

In this model, a system is made up of M processors and a parallel job consists of one process per processor [2]. All processes are dispatched simultaneously at the beginning of the execution. At the end of the execution, the process synchronizes to compose the final work and repeats cyclically. Ideally, the parallel job process should be the only workload assigned to the processor. However, some operating system-specific jobs need to run on all processors for the correct operation of the system, such as the periodic scheduler tick, critical kernel threads, or others. In this scenario, the scheduling decisions of each local processor have a significant impact on the response time of a parallel job. These delays caused to a parallel workload by OS activities running on the same processor(s) is named Operating System Noise. The Operating System Noise measured by osnoise is a generalized version of the classical Operating System Noise. Indeed, it is defined as all the time spent by a CPU executing instructions not belonging to a given application task assigned to that CPU while the task is ready to run. This generalizes the usual interpretation of OS noise, in the sense that it does not include only OS-related activities and overheads, accounting also for the time used by any interfering computational activity, not limited to the OS but also from regular user-space threads. This extended definition gives room for an interesting link between OS noise, a metric from the HPC domain, and the high-priority interference commonly considered in real-time systems theory. Linux has four main execution contexts, which can take control of the CPU where a given workload is running. These are non-maskable interrupts (NMIs), maskable interrupts (IRQs), softirqs (deferred IRQ activities), and threads [18]. Interrupts are managed by the interrupt controller, which queues and dispatches multiple IRQs and an NMI for each CPU. The NMI handler is the highest-priority activity on each CPU, it is non-maskable, and hence it is capable of preempting IRQs and threads. IRQs, in turn, are able to preempt threads and softirqs, unless they have been temporarily disabled within critical sections of the kernel. Softirq is a software abstraction in the standard kernel configuration and runs after IRQ execution, preempting threads. Finally, threads are the task abstraction managed by Linux schedulers. Another source of noise that must be taken into account is hardware-induced noise. Hardware noise can be a side effect of hardware stalls caused by shared resources, as occurring in hyperthreading enabled processors or execution contexts with a higher priority than the OS, like SMIs [26].

2.1.1 The osnoise workloads threads

The osnoise workload threads used for measurements work on a per-CPU basis. By default, osnoise creates a periodic kernel thread on each CPU. The kernel thread can be assigned to any Linux scheduler, such as SCHED_DEADLINE [29], SCHED_FIFO, SCHED_RR, or CFS . Each thread runs for a pre-determined amount of runtime. The primary purpose of the workload thread is to detect the time stolen from its execution, which is considered OS noise. Each osnoise thread works by reading the time in a loop. When a gap between two consecutive readings greater than a given tolerance threshold is detected, a new noise sample is collected. After run-time microseconds have elapsed since the first read of the current period, the workload reports a summary of the OS noise faced by the current activation.

```
[root@f35 tracing]# cat trace
# tracer: osnoise
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth                            MAX
#                              || /                                             SINGLE     Interference counters:
#                              ||||               RUNTIME      NOISE   % OF CPU  NOISE    +-----------------------------+
#           TASK-PID      CPU# ||||   TIMESTAMP    IN US       IN US  AVAILABLE  IN US      HW    NMI    IRQ   SIRQ THREAD
#              | |         |   ||||      |           |             |    |            |      |      |      |      |      |
           <...>-859     [000] ....    81.637220: 1000000        190  99.98100       9     18      0   1007     18      1
           <...>-860     [001] ....    81.638154: 1000000        656  99.93440      74     23      0   1006     16      3
           <...>-861     [002] ....    81.638193: 1000000       5675  99.43250     202      6      0   1013     25     21
           <...>-862     [003] ....    81.638242: 1000000        125  99.98750      45      1      0   1011     23      0
           <...>-863     [004] ....    81.638260: 1000000       1721  99.82790     168      7      0   1002     49     41
```

Figure 2: osnoise summary output from ftrace interface

The osnoise summary reports:

- RUNTIME IN US, that is, the amount of time in µs in which osnoise looped reading the timestamp.
- NOISE IN US, that is, the total amount of noise in µs observed in the associated run-time.
- PERCENTAGE OF CPU AVAILABLE, i.e. the percentage of CPU available to the osnoise thread in the measuring period.
- MAX SINGLE NOISE IN US, i.e. the longest observed occurrence of noise in µs during the runtime.
- The interference counters: for each type of interference among the classes NMI, IRQs, softirqs, and threads, osnoise maintains an interference counter that is increased in correspondence of an entry event of activity of that type.

2.1.2 The osnoise tracing features

The tracepoints are one of the key pillars of the Linux kernel tracing. The tracepoints are points in the kernel code where it is possible to attach a probe to run a function. They are most commonly used to collect trace information. These callback functions collect the data, saving them to a trace buffer. The data in the trace buffer can then be accessed through a trace interface. The osnoise tracer leverages the current tracing infrastructure in two ways. It adds probes to existing tracepoints to collect information and adds a new set of tracepoints with pre-processed information. Linux already has tracepoints that intercept the entry and exit of IRQs, softirqs, and threads. Osnoise attaches a probe to all entry and exit events and uses it to: 1) account for the number of times each of these classes of tasks added noise to the workload; 2) compute the value of the interference counter used by the workload to identify how many interferences occurred between two consecutive reads of the time; and 3) compute the execution time of the current interfering task. At the exit probe of each of these interference sources, a single osnoise tracepoint is generated, reporting the execution time of the task noise. In addition to the tracepoints and the summary at the end of the period, the osnoise workload emits a tracepoint whenever a noise is identified. This tracepoint informs about the noise observed through workload and the amount of interference that occurred between the two consecutive reads, as shown in Figure 3.

```
    sleep-5842   [003] d.... 203398.433413: thread_noise:     sleep:5842 start 203398.433217481 duration 195472 ns
    bash-5802    [003] d.... 203398.433829: thread_noise:     bash:5802 start 203398.433413330 duration 415172 ns
    sleep-5843   [003] d.h.. 203398.434022: irq_noise:        local_timer:236 start 203398.434016335 duration 5627 ns
    sleep-5843   [003] d.... 203398.434629: thread_noise:     sleep:5843 start 203398.433829263 duration 793261 ns
osnoise/3-4417   [003] ..... 203398.434631: sample_threshold: start 203398.433215747 duration 1414624 ns interference 4
```

Figure 3: osnoise sample_threshold: summary tracepoint of osnoise

2.2. Virtualization and KVM

Virtualization is a process that allows more efficient use of physical computer hardware. Virtualization introduces a layer called a hypervisor/VMM [38] between the underlying hardware and the operating systems running on top of it. This layer allows the hardware elements of a single computer, such as processors, memory, storage, and more, to be divided into multiple virtual computers, commonly called virtual machines (VMs). Each virtual machine runs its own operating system (OS) and behaves like an independent computer, even though it runs on just a portion of the actual underlying computer hardware. Before further discussing virtualization, it is necessary to discuss a concept called "protection rings" [48]. In computer science, various hierarchical protection domains/ privileged rings exist. These are the mechanisms that protect data or faults based on the security enforced when accessing resources in a computer system.
Ring 0 is the level with the most privileges and interacts directly with physical hardware, such as the CPU and memory. Most general-purpose systems use only two rings: ring 0 and ring 3. From an operating system's point of view, Ring 0 is called the kernel mode/supervisor mode, and Ring 3 is the user mode.

Operating systems, such as Linux, use the supervisor/kernel and user mode. The user mode can do almost nothing to the outside world without calling on the kernel or without its help, due to its restricted access to memory, CPU, and I/O ports. The kernels can run in privileged mode, which means that they can run on ring 0. To perform specialized functions, the user mode code must perform a system call to the supervisor mode or even to the kernel space, where a trusted code of the operating system will perform the needed task and return the execution back to the user space. In short, the operating system runs in ring 0 in a normal environment.The hypervisor/Virtual Machine Monitor (VMM) needs to access the memory, CPU, and I/O devices of the host. Since only the code running in ring 0 is allowed to perform these operations, it needs to run in the most privileged ring, which is ring 0, and has to be placed next to the kernel. The virtual machine's operating system must reside in Ring 1. An operating system installed in a virtual machine is also expected to access all resources, as it is unaware of the virtualization layer; To achieve this, it has to run in Ring 0 similar to the VMM. Due to the fact that only one kernel can run on Ring 0 at a time, the guest operating systems have to run in another ring with fewer privileges or have to be modified to run in user mode. To do so, there are two kinds of virtualization: full virtualization and paravirtualization.
In full virtualization [24], privileged instructions are emulated to overcome the limitations arising from the guest operating system running in Ring 1 and the VMM running in Ring 0. It relies on techniques, such as binary translation, to trap and virtualize the execution of certain sensitive and non-virtualizable instructions. That being said, in binary translation, some system calls are interpreted and dynamically rewritten. The previous diagram depicts how the Guest OS accesses host computer hardware through Ring 1 for privileged instructions and how unprivileged instructions are executed without the involvement of Ring 1. With this approach, the critical instructions are discovered (statically or dynamically at run-time) and replaced with traps into the VMM that are to be emulated in software. In paravirtualization [24], the guest operating system needs to be modified to allow those instructions to access Ring 0. In other words, the operating system needs to be modified to communicate between the VMM/hypervisor and the guest through the "backend" (hypercalls) path. Paravirtualization is a technique in which the hypervisor provides an API and the OS of the guest virtual machine calls that API which requires host operating system modifications. The protected instruction calls are exchanged with the API functions provided by the VMM.
However, the latest and most efficient kind of virtualization that has been proposed is hardware-assisted virtualization [21]. Intel and AMD independently created new processor extensions of the x86 architecture, called Intel VT-x and AMD-V, respectively. With hardware-assisted virtualization, the guest operating system for certain instructions has direct access to resources without any emulation or OS modification. Hardware-assisted virtualization greatly improves the performances; indeed, the number of traps caused by system calls and interrupts is reduced. Traps are performed only during sensitive instructions, but not all privileged instructions are sensitive; therefore, a number of privileged instructions that are not sensitive do not need to trap to the hypervisor differently from before. Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two types of VMX operation: VMX root operation and VMX non-root operation. In general, a VMM will run in VMX root operation and guest software will run in VMX non root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two types of VMX transitions. Transitions into VMX non-root operation are called VM entries, while transitions from VMX non-root operation to VMX root operation are called VM exits.

2.2.1 KVM

Before diving into the virtualization technology of Linux, some definitions are needed about the different types of hypervisors. If the VMM/hypervisor runs directly on top of the hardware, it is generally considered to be a Type 1 hypervisor. If there is an operating system present and if the VMM/hypervisor operates as a separate layer, it will be considered as a Type 2 hypervisor. A Type 1 hypervisor interacts directly with the system hardware; it does not need any host operating system. Type 1 hypervisors are also called bare metal, embedded, or native hypervisors. Type 2 hypervisor is on top of the operating system, allowing numerous customizations. Type 2 hypervisors are also known as hosted hypervisors. The Kernel Virtual Machine is the Linux module that allows to use the hardware-assisted virtualization provided by Intel to easily create and manage virtual machines. KVM turns the Linux kernel into a hypervisor. However, as a standard kernel module, it benefits from the changes to the standard kernel (scheduler, memory support, etc.). For I/O emulations, KVM uses a utility software, QEMU, which performs hardware emulation. It indeed emulates the processor and a long list of peripheral devices, disk, network, and so on, building a complete virtual hardware on which the guest operating system can be installed. Each vCPU is just a thread in KVM, which means that it is scheduled as every other Linux task by the Linux scheduler(s). At the same time, each virtual machine is treated as a simple process by QEMU. Hardware-assisted virtualization is a type of virtualization which, using specific instructions in the ISA, allows the Guest OS to directly access the host hardware, without any virtualization. KVM is a type 1 hypervisor that allows the use of this set of new instructions introduced by hardware-assisted virtualization. QEMU is a type 2 hypervisor, which allows one to perform hardware virtualization. KVM is

providing the way to access the new instructions, which allow one to directly run the guest OS into the host hardware, but it needs some hypervisor where the guest OS can reside, and this is what QEMU is used for. KVM provides an interface to the userspace for using the CPU virtualization. This includes functions that userspace can call for "create CPU", "run CPU", etc. For a full virtual machine, userspace code is needed, and this is QEMU. Therefore, guest code executes natively, apart from trap'n emulate instructions. Performance-critical or security-critical operations are handled in the kernel; examples are mode transitions or shadow MMU. I/O emulation and management are handled in user space with QEMU. As already introduced, Virtual CPUs (vCPUs) on a virtual machine are Linux threads, scheduled like other tasks. This results in interruptions from other virtualized tasks and noise from the host. Additionally, when the guest performs sensitive instructions, the hypervisor takes control of the processor, introducing overhead. This, combined with host interruptions from higher-priority contexts, defines virtualization-induced noise. Each time a vCPU is kicked off the processor in order to execute sensitive instructions, or other tasks take control of the CPU, a vmexit is said to happen. An important division to define is the one between lightweight vmexits and heavyweight vmexits. The heavyweight vmexits are all the vmexits that need access to the devices, while the lightweight ones are all the other kinds of vmexits. Lightweight vmexits are handled directly by the hypervisor (KVM).
The handling of a lightweight vmexit is extremely fast with respect to its counterpart. Indeed, heavyweight vmexits are needed for instructions that request access to devices and, therefore, need to pass through QEMU.

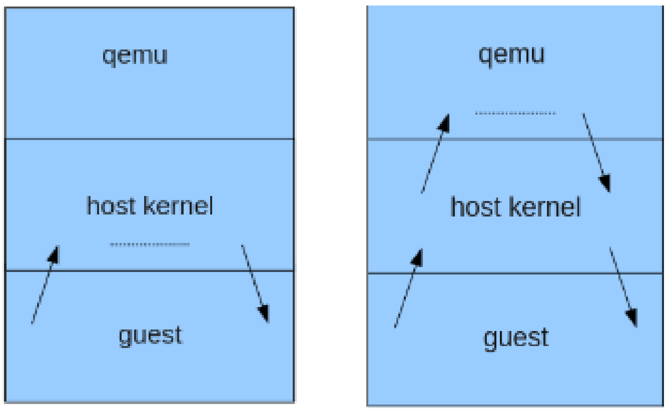


Figure 4: Lightweight vmexit (left): handled only by KVM, heavyweight vmexit (right): requires access to QEMU

3. Related Work

In HPC, a common method to measure OS noise is running micro-benchmarks with known durations and comparing the expected and actual processing times. The fixed-work quantum (FWQ) [43] and fixed-time quantum (FTQ) [31] benchmarks are examples. Both run on each core within a single node. FWQ repetitively performs a fixed amount of work (the work quanta), measuring the time necessary to complete that work. FTQ repetitively works for a fixed amount of time (the time quanta), measuring the amount of work that is performed. Both repeat the measurements multiple times. For FTQ, the work performed is very simple: increment a variable repeatedly. For FWQ, there are three work choices. The default is the same as the FTQ. Otherwise, it can output a time file, one per thread, or two files (per thread), one with the amount of work performed (one per line), the other with the time quantums (one per line). Hensbergen [28] pioneered the analysis of virtualization's impact on HPC applications, using the FWQ benchmark to assess noise levels experienced when running on the IBM research hypervisor, rHype [40]. The workload consisted of measuring the time to execute an empty loop. The experiments were performed first on a single-processor server under a traditional Linux kernel, and then the same server was used to run the micro-benchmark in a stand-alone application kernel running under the rHype hypervisor. The results obtained show the number of milliseconds it took to complete the fixed workload, showing a 20% increase in the time needed to complete the workload when running under the hypervisor. The problem with said output is that it highlights performance degradation due to virtualization noise, leaving the user to inspect the system to come up with the precise causes.
Younge et al. [3] have provided an in-depth analysis of some of the most commonly accepted virtualization technologies today, from feature comparison to performance analysis, focusing on applicability to high-performance computing environments using FutureGrid resources [25]. A total of four computing nodes were assigned for the experiments. Three of the four nodes were installed with different hypervisors; Xen, KVM, and VirtualBox, and the fourth node was left as is to act as a control for native bare metal performance. The performance com-

parison of each virtual machine has been based on two well-known industry standard performance benchmark suites: HPCC [37] and SPEC [19]. These two benchmark environments are recognized for their reproducible and standardized results in the HPC community. The two benchmarks provide a means to stress and compare the processor, memory, interprocess communication, network, and overall performance and throughput of a system. HPCC benchmarks are designed to test system performance at multiple levels. It consists of seven different tests. To cite the most used:

- HPL - The Linpack TPP benchmark measures the floating point rate of execution for solving a linear system of equations. This benchmark is perhaps the most important benchmark within HPC today, as it is the basis for the evaluation of the Top 500 list.
- FFT - Measures the floating point rate of execution of a double precision complex one-dimensional Discrete Fourier Transform (DFT).

The results obtained from these benchmarks provide an unbiased performance analysis of the hypervisors. SPEC benchmarks are the other major standard for evaluating benchmarking systems. Younge et al. started the comparison from the High Performance Linpack (HPL) benchmark, the de facto standard for comparing resources. The native case is capable of around 73.5 Gflops, which, as the authors say, without optimizations achieves 75% of the theoretical peak performance. Xen, KVM and VirtualBox perform at 49.1, 51.8 and 51.3 Gflops, respectively, when averaged over 20 runs. Next, they tested another key benchmark within the HPC community: Fast Fourier Transforms (FFT). In this case, performance across all hypervisors is roughly equal to bare metal performance, a good indication that HPC applications may be well suited for use on VMs. The authors then moved on to the SPEC benchmarks. The combined performance of the 11 applications executed 20 times yields the native testbed with the best performance at a SPEC score of 34465. KVM performance comes close with a score of 34384, which is so similar to native performance that most users will never notice the difference. Xen and VirtualBox perform considerably slower, with scores of 31824 and 31695. Therefore, the study has evaluated the viability of virtualization within HPC, concluding that it is indeed possible, with KVM taking a clear lead by almost matching native speeds.
A similar approach has been carried out by Beserra et al. [17], who used the HPC Challenge Benchmark (HPCC) to evaluate the performance of the processor, RAM, interprocess communication, and network communication with different hypervisors. They have also concluded that KVM in Para virtualization mode has a similar performance to that of a native cluster.
These HPCC benchmarks focus primarily on performance metrics such as operations per second (e.g., FLOPS). They allow the measurement and stress of specific components of a system, such as the processor, memory, interprocess communication, or the network. However, they are capable of highlighting performance issues, without detailed root cause explanations. Kudryavtsev et al. [6] have shown that virtualization noise is one of the main causes of performance degradations highlighted by benchmark suites such as the HPC Challenge. Therefore, osnoise could be used as a complementary tool to check for the causes of these performance deficits.
In the Real-Time community, tools like oslat [49] and sysjitter [50] are the industry standard to measure OS noise by running a thread on each CPU and tracking intervals where the thread is not running due to OS activities. Specifically, oslat runs a CPU-bound busy loop that continuously reads the time. Anything that interrupts the busy loop shows up as a gap. This kind of workload emulates a virtualized RAN workload [16] in a real-world scenario, such as a polling thread using the Data Plane Development Kit (DPDK) [39] to read and write packets to the NIC. Therefore, it has been used to demonstrate equivalent performance for Radio Access Network (RAN) workloads in VMware vSphere and bare metal [34]. However, both oslat and sysjitter quantify the latency of the system, without adding any information on the causes of the latency in case of an increment moving from the bare-metal to the virtualized case.

4. Approach

Osnoise has been extended on both the guest and host side. From the guest extension, it measures noise from the hypervisor and the host, indicating how long the virtual machine was taken away from the processor to execute the hypervisor and higher-priority tasks. However, within the guest, you cannot determine the specific events that caused the interruptions. To address this, osnoise was extended for host-side execution to identify these events, providing a comprehensive understanding of virtualization-induced noise, including numerical values and their root causes. The extensions are presented in conjunction with the evaluation methodology.

4.1. Guest Side

In the vanilla version of the tool, noise detection checks for changes in the number of interferences experienced by the osnoise thread between iterations. If this value remains unchanged, it signifies hardware-induced noise without other task preemptions. However, in virtual machine environments, the virtual CPU executing the osnoise thread could be preempted by the hypervisor or the host to execute higher priority tasks. The guest-side extension of the tool we introduced distinguishes this virtualization-induced noise from hardware-related noise. It does so reading from a shared data structure between the host and the guest. More specifically, KVM allows communication between the guest and the host using Model Specific Registers [12], control registers in the x86 system architecture used for debugging, program execution tracing, computer performance monitoring, and toggling certain CPU features. The MSR used to extend the tool is the MSR_KVM_STEAL_TIME: a 64-byte alignment physical address of memory area in guest RAM. This memory holds a copy of the following structure:

```
struct kvm_steal_time {
      __u64 steal;
      __u32 version;
      __u32 flags;
      __u8  preempted;
      __u8  u8_pad[3];
      __u32 pad[11];
}
```

whose data will be filled in by the hypervisor periodically. Only one write, or registration, is needed for each vCPU. The interval between updates of this structure is arbitrary and implementation dependent. The hypervisor may update this structure at any time it sees fit. The guest is required to ensure that this structure is initialized to zero. The most important field of this data structure is the steal one. In fact, it allows the guest to read the amount of steal time accumulated since it was started. The steal time is defined as the time that a virtual CPU is unscheduled from the physical processor due to host preemption. Therefore, it is a crucial metric to understand the time spent by the physical processor to execute other higher-priority tasks, measuring the amount of noise the guest is experiencing. As a first step, osnoise has been modified to read from this shared data structure, when running on top of a vCPU, to compute the amount of steal time experienced at each iteration of the osnoise thread. It does so using functions provided by the guest-code of KVM. To compute the steal time experienced by the osnoise thread at a certain iteration of the loop, the approach adopted is the same as used by the vanilla version of osnoise to compute the noise at a certain iteration. As shown in the snippet of code below, two variables have been defined: one storing the steal time accumulated since the previous iteration and one used to store the steal time accumulated at this iteration. At this point, the steal time experienced in this iteration is simply the difference between the two variables. Finally, the value is added to the variable that stores the overall steal time experienced by the osnoise thread since it started running.

```

// to store steal time values
 u64 last_steal, steal;

 // steal time experienced at i-th iteration
 u32 is_steal_time;

 // steal time experienced since osnoise started
 u64 steal_time;

do {

        ...

        steal = read_steal();

        ...

        is_steal_time = steal - last_steal;

        ...

        steal_time += is_steal_time;

        ...

        last_steal = steal;

} while (1);
```

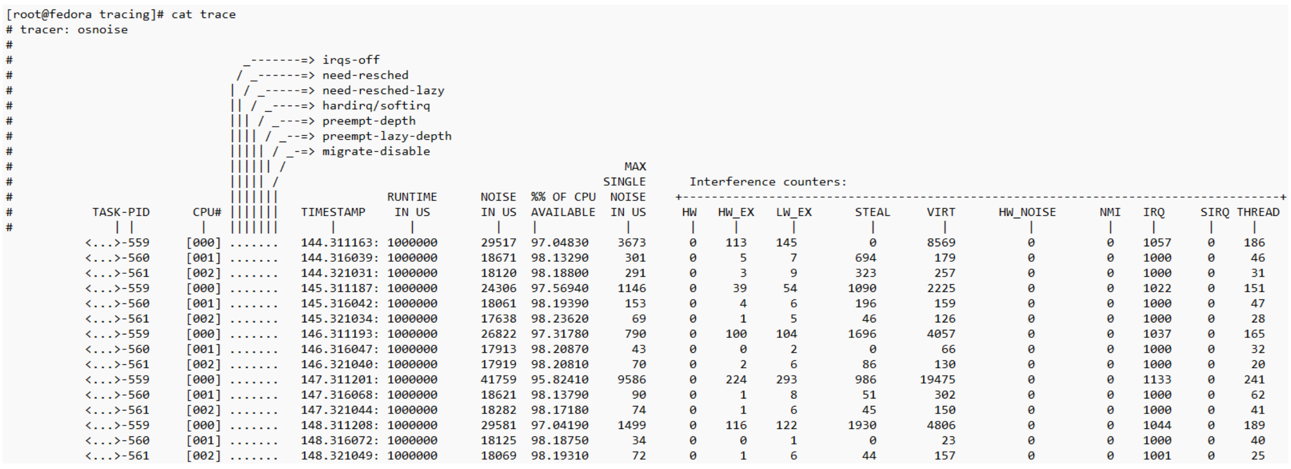
The virtualization noise experienced by a running workload, such as the osnoise thread, is not limited to the time the vCPU where the workload is executing spends de-scheduled due to higher priority tasks. Indeed, the hypervisor takes control of the physical processor each time the guest tries to perform sensitive instructions or to restore the virtual machine state when it is removed from the CPU. This kind of noise is not measured by the steal-time metric. To compute this hypervisor noise, the steal time is subtracted from the total noise experienced by the osnoise thread. Any time not categorized as steal time or hypervisor noise represents the time spent on hardware-related tasks. Therefore, it is necessary to increase the hardware noise with the noise measured during the current iteration only when the noise is not due to the host or the hypervisor. As explained in the background section, in virtualization, when the hypervisor or host takes control of the physical processor, it triggers a vmexit event. KVM distinguishes between a lightweight vmexit handled by KVM itself and a heavyweight vmexit, typically caused by device accesses and involving QEMU intervention, introducing more latency. A vmexit, either lightweight or heavyweight, is a necessary condition for a descheduling of the virtual machine, which also means an increase in the steal time. In fact, each time the virtual machine is descheduled from the CPU, the hypervisor needs to perform tasks such as saving and restoring the state of the virtual machine. Therefore, the kvm_steal_time struct has been modified, in order to include two counters, one for the lightweight vmexits and another for the heavyweight ones, which are incremented each time each of these two types of exits occur.

```

struct kvm_steal_time {
      __u64 steal;
      __u32 version;
      __u32 flags;

      // vmexits counters
      __u32 count[2];
      __u8  preempted;
      __u8  u8_pad[3];

      // reduced by 2
      __u32 pad[9];
}
```

As for the steal time, these two new counters are used to compute the number of lightweight and heavyweight vmexits that have occurred since the osnoise thread has started running on the vCPU. More importantly, they are used to ensure that the hardware noise is updated only when neither the host nor the hypervisor has introduced noise, as shown in the following code.

```

do {

        ...

        lwexit = read_lexits();
        hwexit = read_hwexits();
        ...

        is_lwexit = lwexit - last_lwexit;
        is_hwexit = hwexit - last_hwexit;
        ...

        if (is_lwexit || is_hwexit) {

            lwexit_count += is_lwexit;
            hwexit_count += is_hwexit;

        }

        else {
            // if number of interference == 0
            if (!interference)
                hw_count ++;
        }
        ...

        last_lwexit = lwexit;
        last_hwexit = hwexit;

} while (1);
```

As a final outcome as shown in Figure, when running osnoise on top of a virtual machine, with the extension provided, it is possible to have a quantitative view of the time the vCPU spents descheduled due to host preemption, identified by the steal time, as well as the time spent running the hypervisor, either KVM or QEMU. Finally, with the two added counters, it is possible to understand the type of vmexits experienced by the guest.



```
[root@fedora tracing]# cat trace
# tracer: osnoise
#
#                                _-------=> irqs-off
#                               / _------=> need-resched
#                              | / _-----=> need-resched-lazy
#                              || / _----=> hardirq/softirq
#                              ||| / _---=> preempt-depth
#                              |||| / _--=> preempt-lazy-depth
#                              ||||| / _-=> migrate-disable
#                              |||||| /                                               MAX
#                              ||||| /                                             SINGLE      Interference counters:
#                              |||||||               RUNTIME      NOISE  %% OF CPU  NOISE    +-------------------------------------------------------------------------------------+
#           TASK-PID      CPU# |||||||   TIMESTAMP    IN US       IN US  AVAILABLE  IN US      HW  HW_EX    LW_EX     STEAL     VIRT       HW_NOISE     NMI    IRQ   SIRQ THREAD
#              | |         |   |||||||       |          |           |    |             |        |     |       |          |         |           |          |     |       |     |
           <...>-559     [000] .......   144.311163: 1000000       29517 97.04830    3673      0    113    145          0       8569          0          0    1057     0     186
           <...>-560     [001] .......   144.316039: 1000000       18671 98.13290     301      0      5      7        694        179          0          0    1000     0      46
           <...>-561     [002] .......   144.321031: 1000000       18120 98.18800     291      0      3      9        323        257          0          0    1000     0      31
           <...>-559     [000] .......   145.311187: 1000000       24306 97.56940    1146      0     39     54       1090       2225          0          0    1022     0     151
           <...>-560     [001] .......   145.316042: 1000000       18061 98.19390     153      0      4      6        196        159          0          0    1000     0      47
           <...>-561     [002] .......   145.321034: 1000000       17638 98.23620      69      0      1      5         46        126          0          0    1000     0      28
           <...>-559     [000] .......   146.311193: 1000000       26822 97.31780     790      0    100    104       1696       4057          0          0    1037     0     165
           <...>-560     [001] .......   146.316047: 1000000       17913 98.20870      43      0      0      2          0         66          0          0    1000     0      32
           <...>-561     [002] .......   146.321040: 1000000       17919 98.20810      70      0      2      6         86        130          0          0    1000     0      20
           <...>-559     [000] .......   147.311201: 1000000       41759 95.82410    9586      0    224    293        986      19475          0          0    1133     0     241
           <...>-560     [001] .......   147.316068: 1000000       18621 98.13790      90      0      1      8         51        302          0          0    1000     0      62
           <...>-561     [002] .......   147.321044: 1000000       18282 98.17180      74      0      1      6         45        150          0          0    1000     0      41
           <...>-559     [000] .......   148.311208: 1000000       29581 97.04190    1499      0    116    122       1930       4806          0          0    1044     0     189
           <...>-560     [001] .......   148.316072: 1000000       18125 98.18750      34      0      0      1          0         23          0          0    1000     0      40
           <...>-561     [002] .......   148.321049: 1000000       18069 98.19310      72      0      1      6         44        157          0          0    1001     0      25
```

Figure 5: Example of output when running osnoise on top of a virtual instance.

As shown in the figure above, new statistics have been added. The LW_EX and HW_EX keep track respectively of the number of lightweight and heavyweight vmexits measured during a certain runtime. The VIRT column measures the time spent executing the hypervisor (either KVM or QEMU) in us, while the STEAL column outputs the time the vCPU has spent descheduled from the physical processor due to higher priority tasks running in the host. Finally, the HW_NOISE column gives the time spent executing hardware tasks (SMIs, etc.).

4.2. Host Side

The guest side extension of the tool provides the numerical values about the noise introduced by the virtualization layer, helping the user to understand the impact that virtualization is having on the overall noise. However, it is not yet possible to identify the events that cause the noise. The host extension of the tool tries to fill this gap, by adding a new tracepoint to the tracing component of osnoise, which, combined with the already present tracepoints, allows the user to identify the events in the host, causing noise in the running guest. The new tracepoint is active only when it is run on the host; indeed, all the events causing noise to the guest are visible only inside the host where the virtual machine is executing. The objective of the tracepoint is to identify each time a vCPU is kicked off the physical processor where it is running. More specifically, the tracepoint output shows the user the following information:

- VM id: ID of the vCPU descheduled.
- Exit CPU: Physical processor from where the vCPU is removed due to vmexit.
- Exit time: Timestamp at which the vmexit happens.
- Entry CPU: Physical processor where the vCPU enters (could be different from the exited CPU due to migration).
- Duration: Overall duration of the vmexit.
- Exit reason: Reason for the vmexit.
- Overhead: Time of vmexit spent executing the hypervisor (KVM/QEMU).

To collect the data provided by the tracepoint, probe functions were attached to the tracepoints already provided by the kvm subsystem in the kernel tracing infrastructure. The two tracepoints of interest are the one tracing each time a certain vCPU exits the physical processor (vmexit) and each time a vCPU enters the same physical processor (vmentry). Next, the probe functions attached to the tracepoints tracing the thread exit and thread entry events from the CPU, already defined by osnoise, were modified. Indeed, even though the vCPU is considered as a simple thread in KVM, the vmexit/vmentry events are considered separately from the thread exit/thread entry ones. This is because when a vmexit happens, before descheduling the vCPU thread from the physical processor, the hypervisor takes control of the CPU. The same goes between the thread entry and the vmentry events.

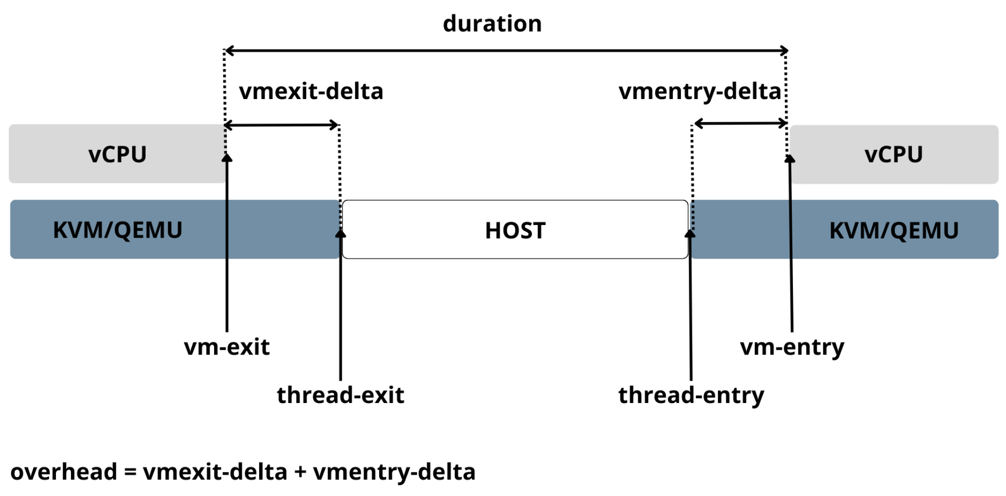


Figure 6: Vmexit with Descheduling.

If the vmexit does not require any descheduling of the vCPU and can be completely handled by the hypervisor, then the thread exit and thread entry tracepoints are never hooked.

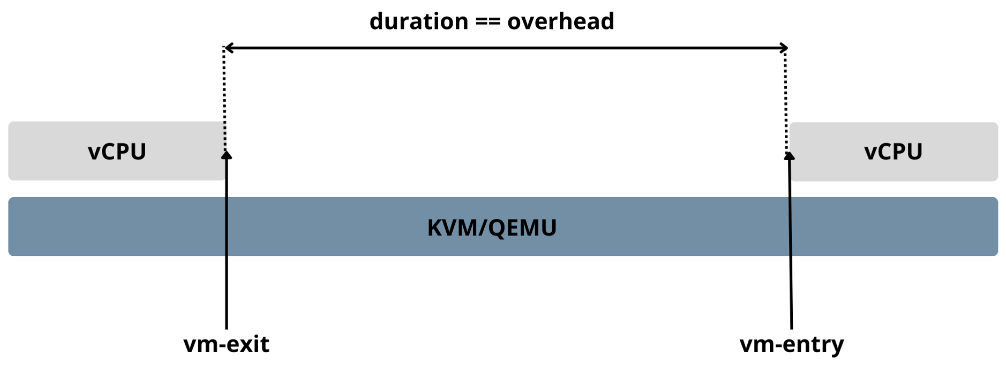


Figure 7: Vmexit without Descheduling.

The overall duration of the vmexit is defined as the difference between the vmentry and vmexit timestamps, while the overhead introduced by the hypervisor is computed as the sum of the two overheads between the vmentry and vmexit. If the vmexit does not require descheduling, then the overhead and duration of the vmexit coincide. In order to hold the information that needs to be printed by the new tracepoint, a struct has been added to the osnoise data.

```

```

```c
struct osn_vm {
    u64 exit_time;
    u64 delta_overhead;
    u64 exit_reason;
    int exit_cpu;
    int not_descheduled;
    struct kvm_vcpu *vcpu;
};
```

When the vmexit tracepoint is triggered, a probe function registers the exit time, the exit reason, and the exit cpu at which it occurred. More importantly, it stores in the vcpu field of the struct defined above, the kvm_vcpu struct passed by the vmexit tracepoint. If vmexit causes a descheduling of the vCPU, the thread exit tracepoint is woken up once the hypervisor has finished running. At this point, delta_overhead variable is set equal to the difference between the exit_time timestamp and the current time at which the thread exit occurs, registering the overhead introduced by the hypervisor between the vmexit and thread exit events. The osn_vm struct, as any struct defined in osnoise, is local to the CPU where the osnoise thread is currently running. However, when the vCPUs are removed from the physical processor, they can possibly be migrated to other cores by the scheduler. Therefore, it is necessary to save the information stored in the newly defined struct in another global data structure, accessible from all the CPUs of the system. To do so, using the kernel hashtable API [32], a global hashtable has been defined, having as key the PID of the vCPU threads and as value the kvm_vcpu struct associated to that PID. When the probe function associated with the thread exit event is called, all the data previously defined are saved in the vcpu variable, which is then added to the global hashtable having as key the PID of the vCPU getting removed. This way, when the vCPU enters the new processor (which can coincide with the one left), using its own PID, it is possible to retrieve the vcpu struct from the hashtable, having access to all the data saved before. When the vmentry tracepoints is hooked up, a probe function computes the overall duration of the vmexit as a difference between the current timestamp and the exit time retrieved from the vcpu variable. Then, the total overhead of the hypervisor is computed adding to the first overhead computed before, the new one given by the time passed between the thread entry and vmentry events. Finally, the new tracepoint is activated, giving in output all the information about the vmexit event.

New tracepoint output example (descheduling)

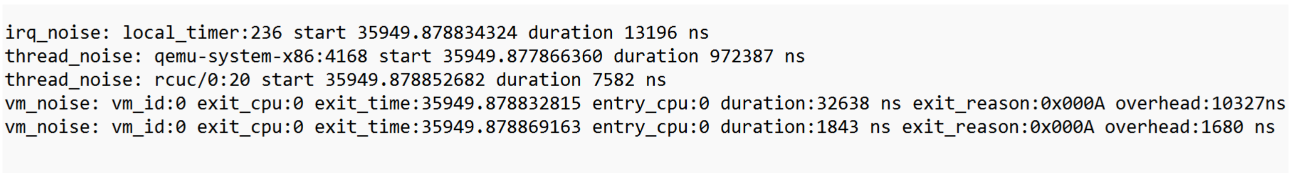
vm_id: 0 exit_cpu: 0 exit_time: 35949.878832815 entry_cpu:0 duration: 326838 ns exit_reason: 0x000A overhead: 10327 ns

As already explained, if the vmexit does not include a descheduling of the vCPU, then the overhead of the hypervisor coincides with the overall duration of the vmexit.

New tracepoint output example (no descheduling)

vm_id: 0 exit_cpu: 0 exit_time: 35949.878869163 entry_cpu: 0 duration: 1843 ns exit_reason: 0x000A overhead: 1680 ns

The tracepoint is intended to be used with the other tracepoints already provided by osnoise, in order to reconstruct the events that happened between the vmexit and vmentry, to better understand the virtualization noise that has been measured by running osnoise on the guest.



```
irq_noise: local_timer:236 start 35949.878834324 duration 13196 ns
thread_noise: qemu-system-x86:4168 start 35949.877866360 duration 972387 ns
thread_noise: rcuc/0:20 start 35949.878852682 duration 7582 ns
vm_noise: vm_id:0 exit_cpu:0 exit_time:35949.878832815 entry_cpu:0 duration:32638 ns exit_reason:0x000A overhead:10327ns
vm_noise: vm_id:0 exit_cpu:0 exit_time:35949.878869163 entry_cpu:0 duration:1843 ns exit_reason:0x000A overhead:1680 ns
```

Figure 8: vm_noise tracepoint allows to reconstruct vmexits events.

As shown in the following figure, the trace output allows reconstructing the events during the vmexit. Indeed, by adding the single durations of each noise, the obtained result matches, except for some negligible differences, the overall duration represented in the vm_noise tracepoint.

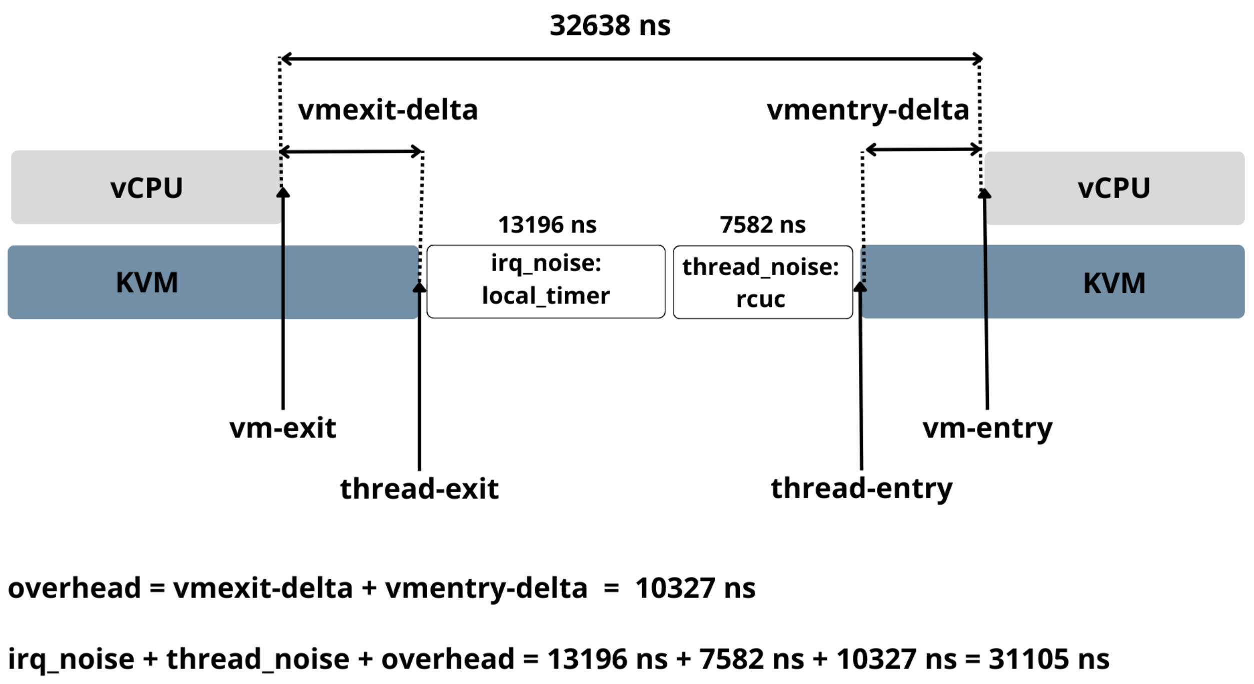


Figure 9: Timeline of the events happened during the vmexit.

The remaining thread noise in Figure 9, is the vCPU itself, running in the core with osnoise, which has preempted the osnoise thread. The last vm_noise instead represent a vm_exit completely handled by the hypervisor, as the overhead and duration have the same values, and in fact it does not have any other interferences traced by osnoise preceding it. Another important thing to notice is that the duration observed via the new tracepoint accounts for 32638 ns, but the total duration obtained by the sum of each contribution (irq_noise, thread_noise and hypervisor overhead) is 31105 ns. The reasons behind are multiple. For example, the overhead added by the tracepoints; the delays added by the hardware to manage context switch and the dispatch of IRQs handlers or the delays caused by cache inlocality after an interrupt [42]. On a lower level, the code that enables the tracing at IRQ context, like making the RCU aware of the current context; and the scheduler call caused by the thread noise.

4.3. Evaluation Methodology

To begin, it is important to highlight the verification process undertaken for each extension of the tool. This verification served as the foundation, ensuring that the output generated by osnoise remained in alignment with the anticipated results. This alignment laid the foundation for the robustness and accuracy of the extended capabilities.
Subsequently, a series of experiments were orchestrated to provide concrete evidence of the extension's validity and its potential to yield substantial benefits in expediting system debugging. These experiments not only served as a means to validate the extension, but also elucidated how it could be practically harnessed to enhance the debugging process.
The experimental framework extended to the ODROID-H2 board, featuring an Intel Celeron J4105 processor with four cores. Within this environment, the system and its virtual instances operated on Fedora Linux 37, integrated with the Linux kernel version 6.2, enhanced with the PREEMPT-RT patchset [23]. Virtual machines, built through QEMU, were endowed with 2 GB of RAM and configured with three cores, optimizing their performance and resource allocation.

4.3.1 Verification

On the guest side, the initial step primarily involved an evaluation of the accuracy of the modified osnoise output. First, a comprehensive verification process was conducted, establishing that the hardware noise measurements within the virtual machine, facilitated by the modified tool, precisely corresponded to the hardware noise levels obtained during the execution of osnoise directly on the host. This correlation was significant, as hardware noise remains unrelated to the realm of virtualization. To dive deeper into the consistency of the output, a secondary run was carried out to scrutinize the various types of vmexits that exert influence on the guest environment. Notably, in a CPU-bound workload such as osnoise, the majority of these exits were expected to fall within the category of lightweight vmexits. To verify the consistency of the tool's extension on the host, first, the overhead introduced by running osnoise has been measured. As a first step, two small virtual machines were run in separate moments, using the kvm-unit-tests suite [1]. These virtual machines were configured to execute two distinct instructions: CPUID and I/O port reading. The former instruction involved the guest performing a series of CPUID instructions [11], which could be efficiently managed by KVM, leading to lightweight vmexits. The latter instruction required the guest to read a 32-bit value from an I/O port, necessitating QEMU emulation and subsequently leading to heavyweight vmexits. With the new introduced tracepoint, the kvm_entry and kvm_exit tracepoints provided by the kvm subsystem are activated each time the vCPU exits or enters the core. This causes an increase in the number of clock cycles needed to perform instructions causing vmexits/vmentry, as the ones performed by the two virtual machines. This happens because the tracepoints's code must be executed. The kvm-unit-tests work by executing the said instructions in loop for a certain amount of time, giving in output the number of clock cycles needed to perform each of the test. Therefore, for both instructions, the number of clock cycles was recorded. As a second step, osnoise was executed directly on the host, first with the virtual machine executing the CPUID instruction, and then with the one reading from the I/O port. The number of clock cycles needed on both tests was then compared with the previous case, in order to understand if the overhead introduced by the tracing was ammissible. As the last part of the verification process on the host side, osnoise was run again, also this time, first with the virtual machine executing the CPUID instruction, second with the I/O port reading. The osnoise tracer was configured to trace both the osnoise and the kvm events. This setup was essential in ensuring that the duration of the vmexit, as measured by the new osnoise tracepoint, closely aligned with the temporal interval elapsed between the vmentry and vmexit events that were traced during the execution of the measured vmexit. This synchronization was needed to verify the accuracy and reliability of the measurement process.

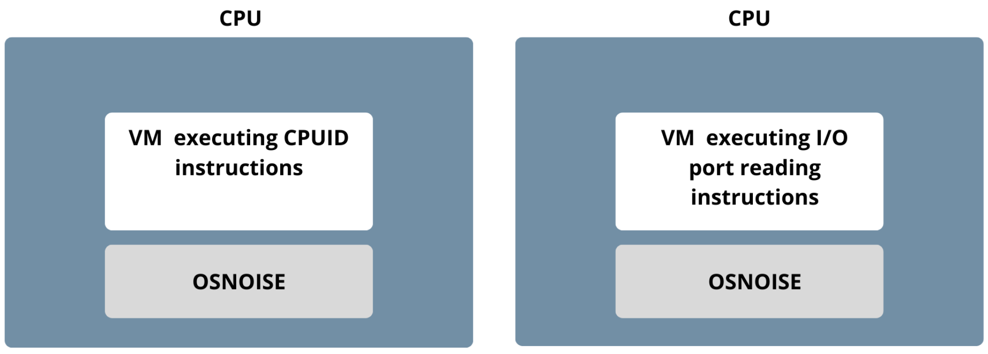

CPU
VM executing CPUID instructions
OSNOISE
CPU
VM executing I/O port reading instructions
OSNOISE

Figure 10: First the a virtual machine executing CPUID instructions (left) then a virtual machine reading from an I/O port (right).

4.3.2 Validation

The tool, in its practical application, has been instrumental in evaluating the feasibility of performing low-latency tasks within a virtualized platform. To do so, as a first step, both hardware and Linux have been configured according to standard practices from both the HPC and the real-time domains.
The hardware is configured to strike a balance between performance and determinism. This setup entails fine-tuning the processor's speed and power-saving parameters, all the while deactivating features that have the potential to introduce hardware-induced latencies. An exemplary instance of these latency-inducing features is the management of system interrupts, notably system management interrupts (SMIs), which are disabled to enhance the system's responsiveness and predictability.
When it comes to configuring the Linux environment, the system is typically organized into an arrangement of isolated and housekeeping CPUs, a practice in HPC systems. Housekeeping CPUs are designated as the processing ground for essential tasks integral to the regular operation of the system. These tasks include

the management of kernel threads responsible for crucial in-kernel mechanisms, such as the management of Read-Copy-Update (RCU) callback threads [36], the execution of kernel threads assigned to deferred work, exemplified by kworkers, and the dispatching of threads by daemons and users. Furthermore, the role of managing general system Interrupt Requests (IRQs) also falls within the responsibility of the housekeeping CPUs [47]. Nevertheless, despite the degree of CPU isolation that Linux currently offers, there remains a need to distribute the housekeeping work between all CPUs. For instance, under certain conditions, the timer IRQ must still be accommodated, and specific kernel activities require the dispatch of kworkers to execute tasks on all available CPUs. This isolation practice is also commonly accepted in the real-time community, in order to implement real-time virtualization [10], [46], [30]. Indeed, the problem with real-time setups in the virtualization context is that priority of tasks inside a VM is not visible to the host and the host cannot identify the vCPU with the highest priority program. Therefore, to solve this issue, vCPU threads need a high real-time priority on the host to guarantee that the real-time applications run when they want. However, the risk is that lower priority tasks coming from the host could starve forever, leading to system deadlock. For this reason, the vCPU threads run on dedicated CPUs, while the host housekeeping is offloaded on housekeeping CPUs. As an initial setup in the validation part, the vCPUs have been run with real-time priority in order to adhere to the best practice adopted in the real-time community. However, performance decreased steadily, with the system blocking due to deadlock, therefore, it has been decided to leave the default priority. This behavior highlighted a problem in the experimental setup that was later confirmed by the osnoise output.
The following figure shows the system appearance in the case in which none of these tunings are applied, nor on the host system, nor on the guest.

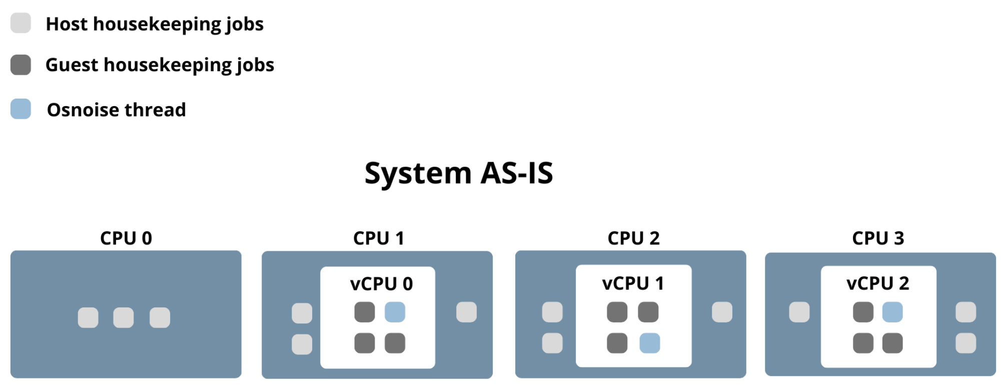
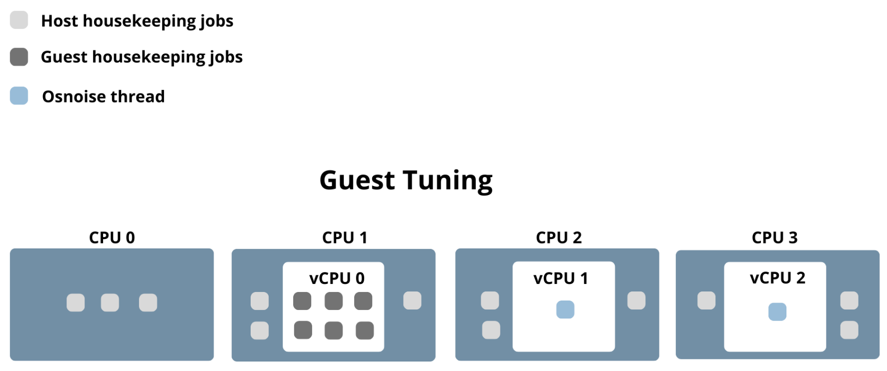


Figure 11: System running without any tuning applied.

Following the tuning practices described above, a division of CPUs was instituted, choosing some as isolated and others as housekeeping sets. The housekeeping CPUs were reserved exclusively for general system tasks, whereas the isolated CPUs were devoted to the exclusive execution of low-latency operations. This allocation of resources was extended to both the host and the guest operating systems, ensuring a comprehensive assessment of the impact of these configurations.
At this juncture, osnoise was executed on the guest, employing various tuning configurations to explore its adaptability to diverse scenarios. Initially, CPU isolation was applied exclusively to the host system, followed by an isolated guest configuration.



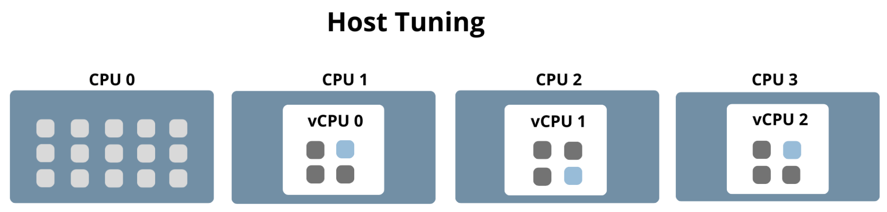


Figure 12: On top the tuning applied solely on the guest, on the bottom the host tuning configuration.

Finally, the most demanding isolation scenario was tested. In this rigorous test, both the host and the guest systems underwent fine-tuning to isolate their respective CPUs. Throughout each experiment, osnoise documented statistics pertaining to virtualization noise.

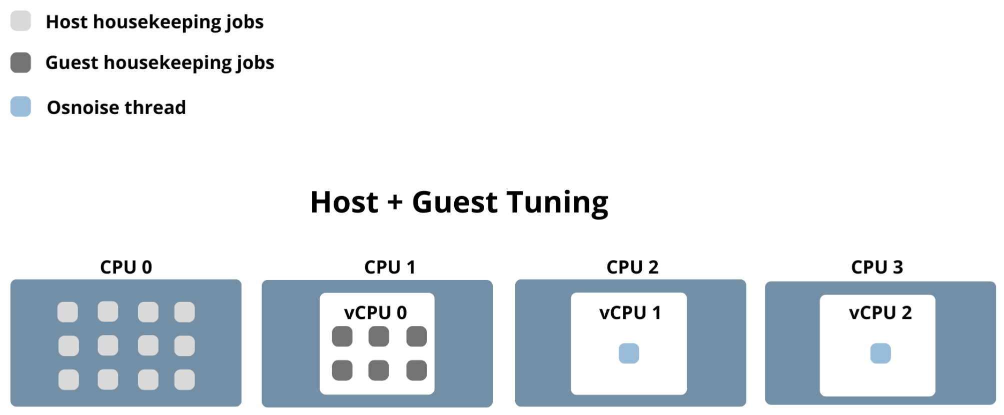


Figure 13: Maximum isolation case: guest and host tunings.

To understand the causes of the noise measured on the guest side, a virtual machine was launched with CPU isolation applied to both the host and the virtual instance. Osnoise was run on isolated vCPUs within the guest, while osnoise was running simultaneously on the host. The new tracepoint was used to collect the events that caused virtualization noise measured in the guest, showing how osnoise is able to provide both quantitative insights into hypervisor/host-induced noise and the underlying causes.

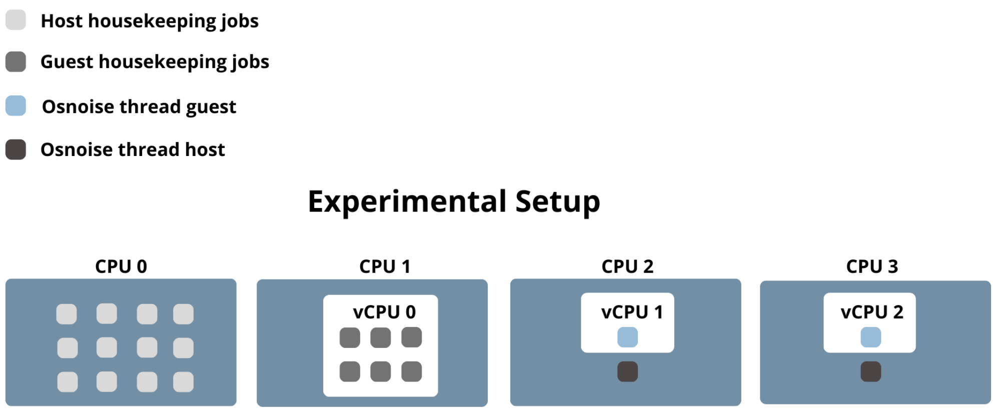


Figure 14: Setup to derive the events causing noise on the osnoise guest thread.

The experimental setup has drawn tuning configurations from both the HPC and real-time worlds. Network Function Virtualization (NFV) is a domain which lays in between HPC and real-time. Indeed, similar to the HPC case, NFV applications receive dedicated resources, including dedicated isolated CPUs. Moreover, latencies of the order of tens of microseconds need to be met, requiring real-time constraints. In NFV, to achieve high throughput, the generic network stack of the operating system is often bypassed, and all network packet processing is done in the user space by a specific process that handles the network flows. To even reduce latency for the handling of new packets, some of these network applications poll the network in a busy-wait fashion, most notably using the DPDK poll mode driver (PMD) [20]. These kinds of workload resemble the onoise thread, which consists of a task reading the time in loop in order to compute noise instances. Osnoise can be used to empirically estimate the high-priority interference faced by an NFV workload running at a given priority on top of a virtual instance. Indeed, the system engineer can set osnoise to run at the same priority, thus exposing the measurement thread to the same sources of noise, and could then use the tool to derive the events that have disturbed the workload if the detected noise exceeds acceptable values. Therefore, osnoise emerges as a novel tool for debugging systems in which these kind of applications need to run. Indeed, differently from industry standard tools such as oslat and sysjitter, osnoise is capable of measuring the main source of noise due to virtualization and to account for the events that have caused such noise.

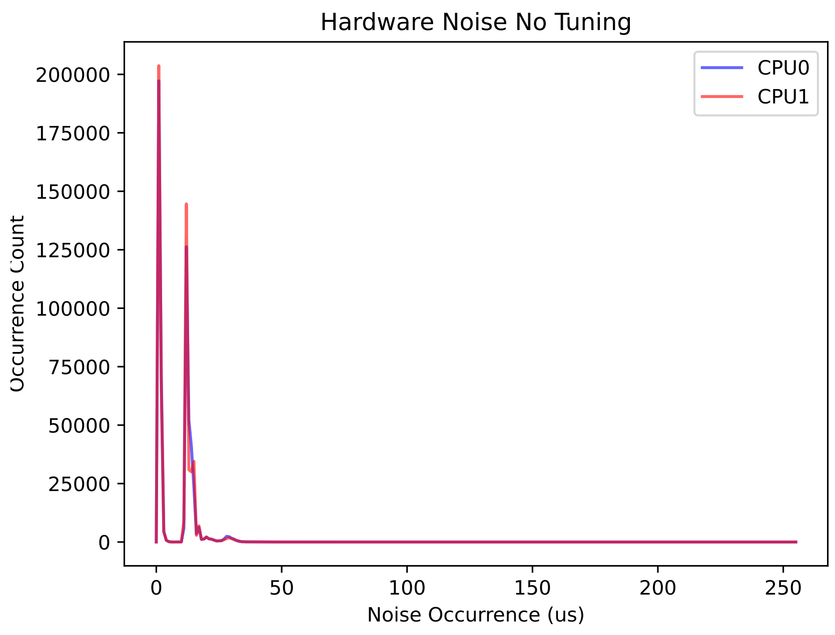
5. Evaluation

The experimental results are presented. First experiments have been performed to verify the correctness of the output provided by osnoise; then the tool has been practically used in order to discover the causes of non-negligible noise, validating its practical utility.

5.1. Verification

5.1.1 Guest Side

The baseline version of osnoise, when run on a virtual machine, showed a significant contribution of hardware noise to the overall one. However, when the extended version of the tool was used, the hardware noise decreased. The reason being that in the vanilla version of the tool, the hardware noise included the noise coming from the virtualization layer. Therefore, with the new changes introduced, the tool is able to distinguish between the noise coming from the virtualization layer and the hardware noise.



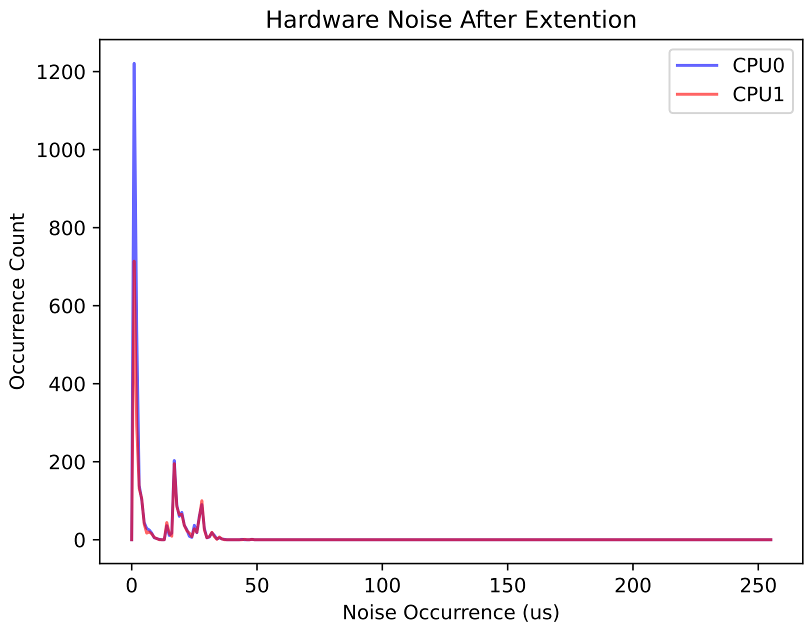
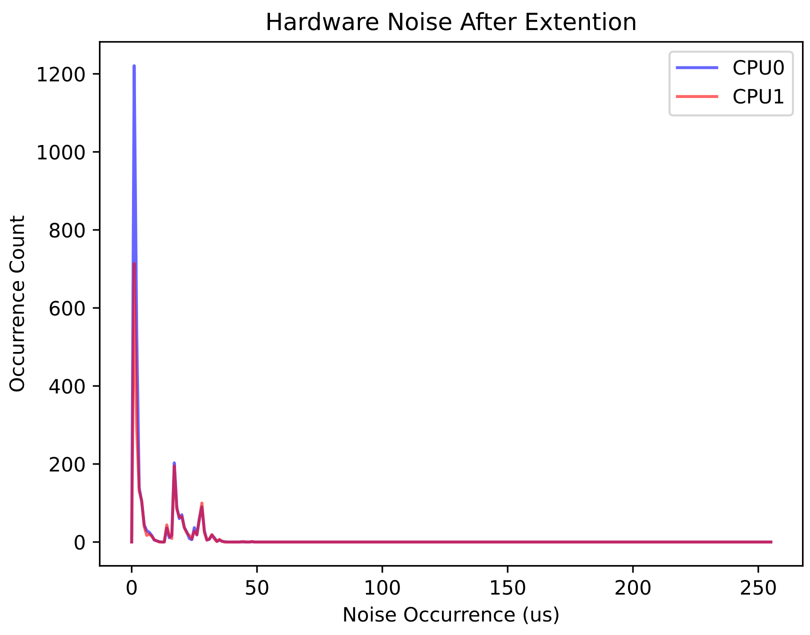


Figure 15: Top: the hardware noise measured before the osnoise extension. Bottom: hardware noise after the extension.

Notably, the hardware noise measured in the guest with the extended version matched that obtained when running osnoise directly on the host. This shows the coherency of the output provided, the hardware noise being not related to virtualization.



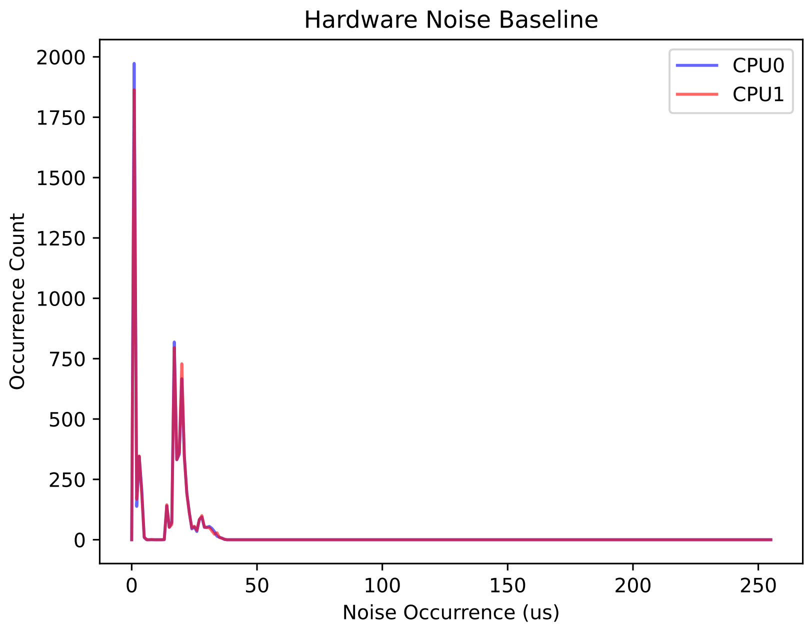


Figure 16: Top: Hardware noise measured on the host (baseline). Bottom: hardware noise on top of the virtual instance measured with the extended osnoise version.

As a second step to verify the correctness of the output, osnoise was utilized to examine the types of vmexits encountered by the guest. Most detected vmexits were of the lightweight variety, which was anticipated, as heavyweight exits associated with device accesses are less common in a CPU-bound workload like osnoise.

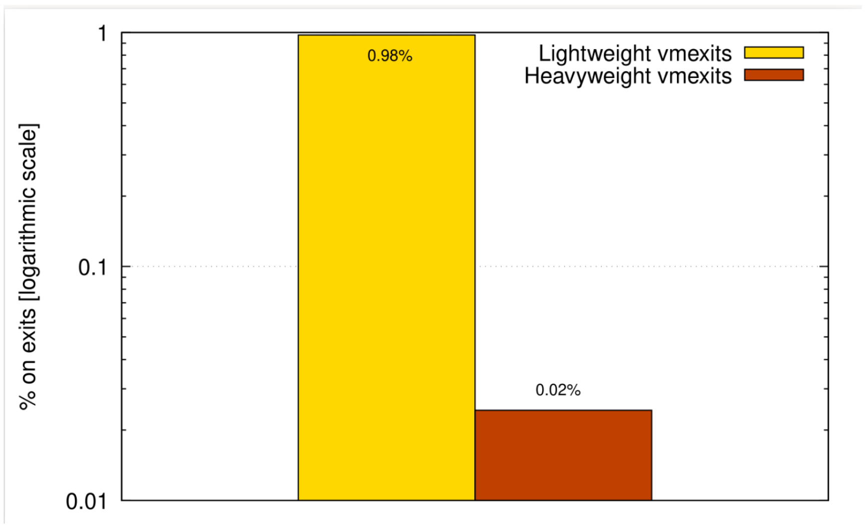


Figure 17: Percentage of Lightweight exits compared with the one of Heavyweight exits.

5.1.2 Host Side

To verify the correctness of the host extension of the tool, the following steps were performed:

- Osnoise was executed directly on the host alongside a virtual machine running the CPUID instruction.
- Osnoise was executed on the host alongside a virtual machine performing I/O port reads.
- The osnoise tracer was configured to trace both osnoise and kvm events.

The output generated by the additional tracepoint aligned with the type of instruction executed in both cases. The CPUID instruction led to lightweight vmexits handled directly by the hypervisor, resulting in a duration of a few microseconds.

```
kvm_exit:       reason CPUID rip 0x4001cd info 0 0
kvm_cpuid:      func 0 idx 0 rax d rbx 756e6547 rcx 6c6576e rdx 49656e69, cpuid entry found
kvm_entry:      vcpu 0 rip 0x4001cf
vm_noise:       vm_id:0 exit_cpu:1 exit_time:3050.313625639 entry_cpu:1 duration:2390ns exit_reason:0x000A overhead:2279ns
```

Figure 18: Osnoise output tracing vm_noise tracepoint and kvm_events after CPUID instruction.

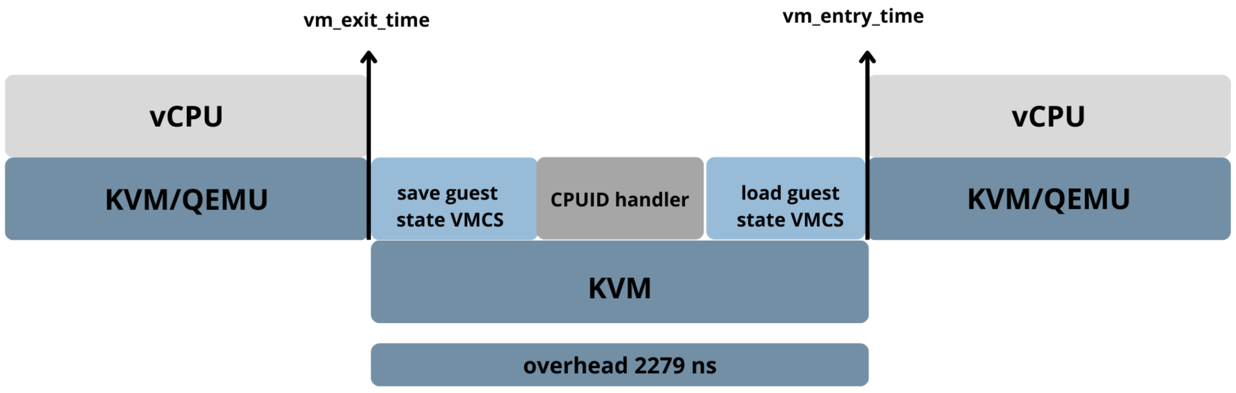


Figure 19: Timeline of events on CPUID lightweight exit.

In contrast, the I/O instructions required hardware emulation by QEMU, causing an increase in the duration of vmexit.

```
kvm_exit:               reason IO_INSTRUCTION rip 0x4052db info 3fd0008 0
kvm_fpu:                unload
kvm_userspace_exit:     reason KVM_EXIT_IO (2)
kvm_fpu:                load
kvm_pio:                pio_read at 0x3fd size 1 count 1 val 0x60
kvm_entry:              vcpu 0 rip 0x4052dc
vm_noise:               vm_id:0 exit_cpu:1 exit_time:3050.3137858566 entry_cpu:1 duration:13334ns exit_reason:0x0001E overhead:13220ns
```

Figure 20: Osnoise output tracing vm_noise tracepoint and kvm_events after I/O port reading instruction.

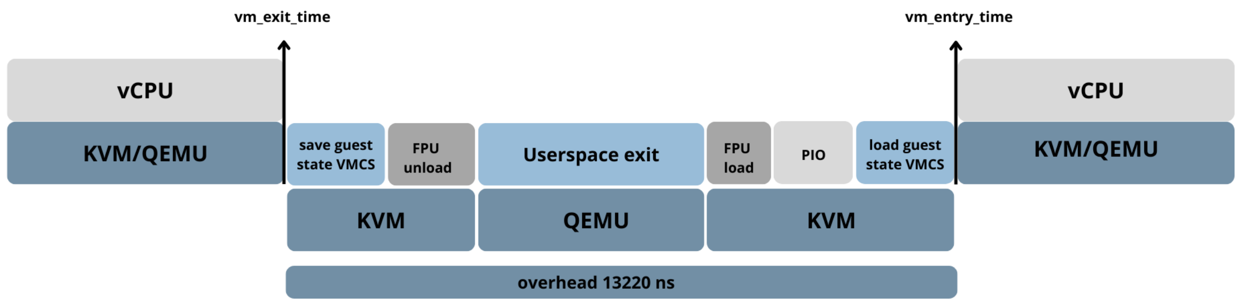


Figure 21: Timeline of events on I/O port reading heavyweight exit.

The two instructions have been repeated several times with the kvm-unit-test in order to check if the results provided by the osnoise tracepoint have always been consistent with the type of vmexit performed.

	n. instruction	average overhead	min. overhead
CPUID	16040	2535.42 ns	2199 ns
I/O	125178	13734.75 ns	9534 ns

Table 1: Statistics about the overhead measured by osnoise for both type of instructions.

More importantly, both with the CPUID and the I/O reading instruction, the duration of the vmexit, as measured by the new osnoise tracepoint, aligned with the temporal interval elapsed between the vmentry and vmexit events traced with the kvm tracing subsystem. This result verify the information provided by the new tracepoint.
Finally, the number of clock cycles needed to perform both the CPUID instructions and the I/O port readings was registered with osnoise running in the background and without it in order to understand if the overhead introduced by the tracing was negligible. The results obtained confirm that running osnoise introduces low overhead, which does not tamper the measurement performed with it.

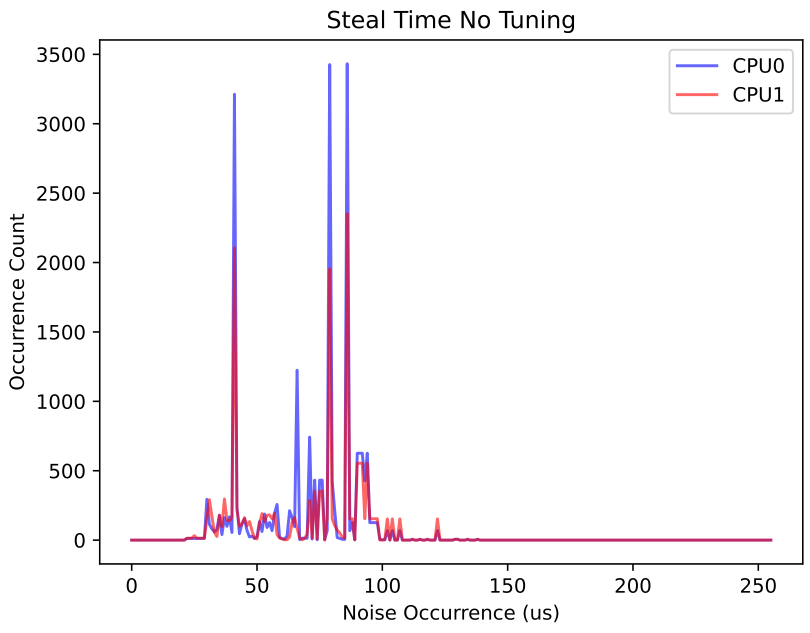
	n. clock cycles (no osnoise)	n. clock cycles (with osnoise)
CPUID	6280	8483
I/O	19942	22631

Table 2: Number of clock cycles detected with kvm-unit-test to perform both instructions, with and without osnoise running.

5.2. Validation

After verifying the coherency of the output, osnoise was used to measure the noise of the virtualization layer and assess the feasibility of running low-latency tasks within a virtual machine. Experiments were conducted

using various tuning configurations as previously described.
Running osnoise with CPU isolation applied on the host led to a significant drop in steal time, indicating minimal descheduling of the guest. Noise caused by KVM and QEMU also decreased, because fewer preemptions require less job by the hypervisor to save the virtual machine state.



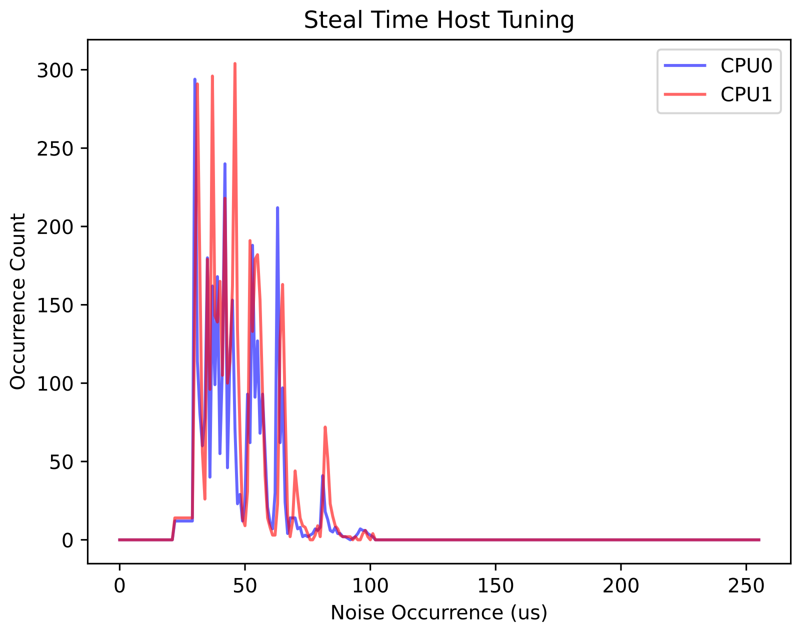
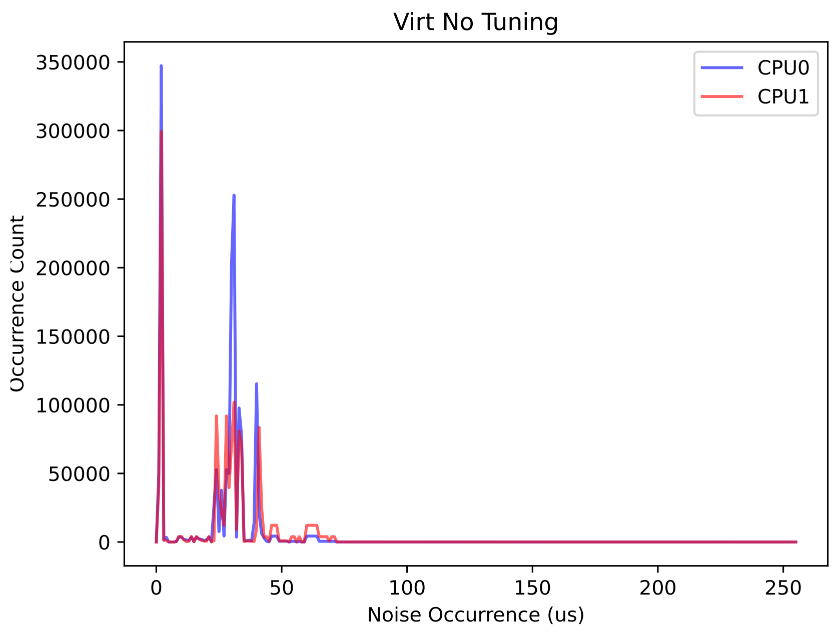


Figure 22: Top: the steal time experienced by the virtual machine before host tuning. Bottom: steal time after host tuning.



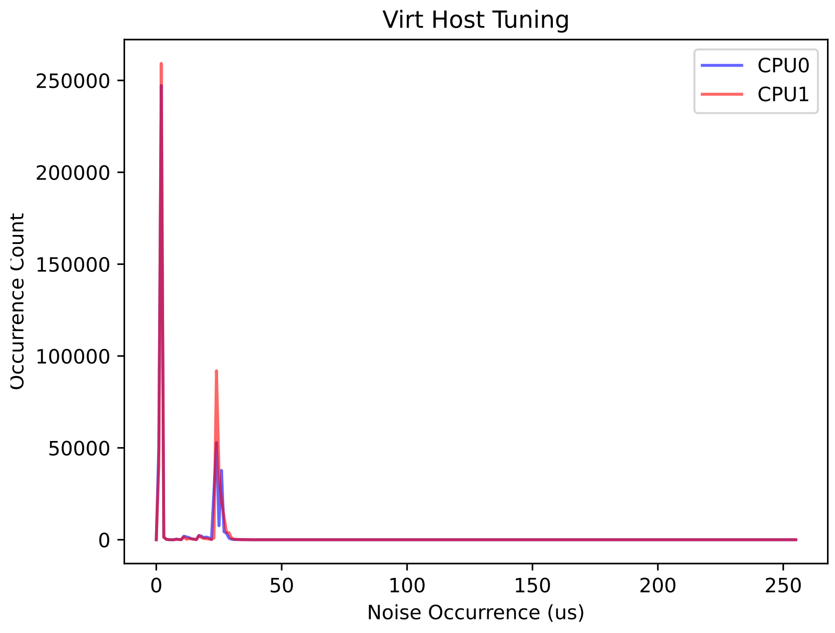
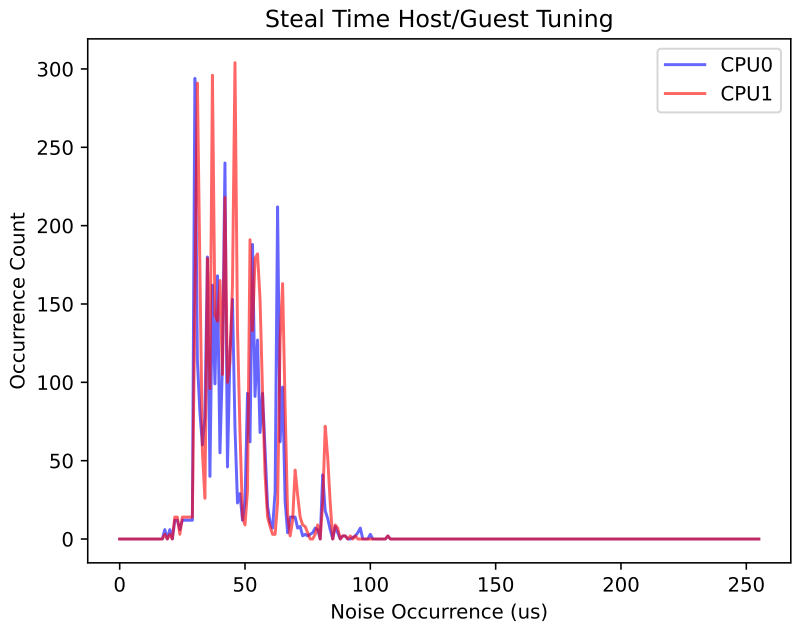


Figure 23: Top: virtualization noise experienced by the virtual machine before host tuning. Bottom: noise introduced by the hypervisor after host tuning.

In the case where CPU isolation was implemented only on the guest side, improvements were modest compared to host tuning. Steal time remained unchanged, while the KVM/QEMU noise decreased slightly.
Tuning both host and guest with CPU isolation achieved the lowest noise levels. Nevertheless, even in this extreme isolation scenario, osnoise's statistics indicated that both steal time and hypervisor noise exceeded acceptable values for low-latency tasks.



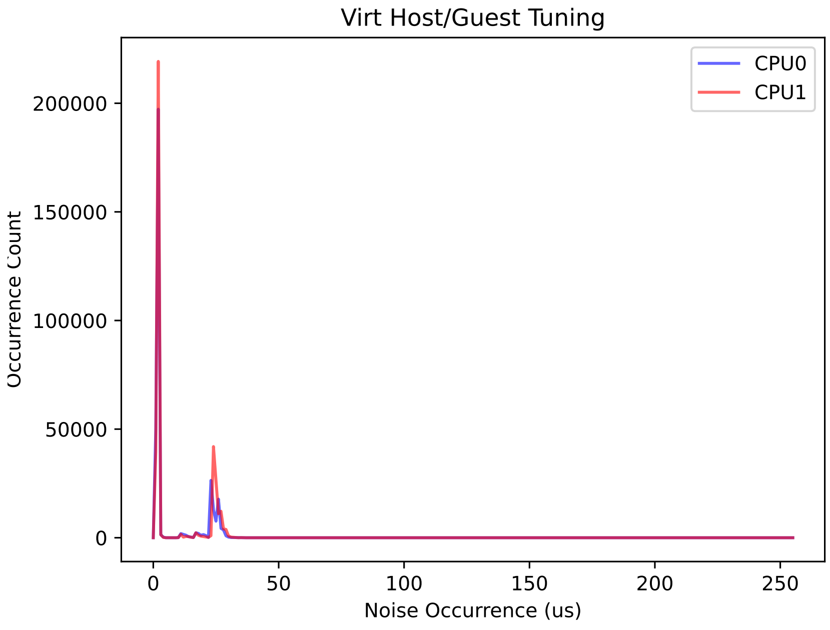
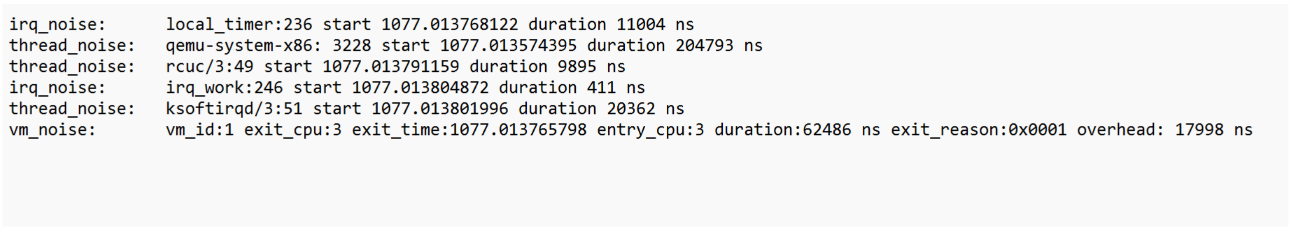


Figure 24: Top: the steal time experienced by the virtual machine with the maximum tuning applied. Bottom: Virtualization noise after having tuned both host and guest.

The analysis of virtualization noise, previously measured by running osnoise on the guest, was carried out by replicating the experimental setup used to test the guest extension of the tool. The following steps were adopted:

- A virtual machine was launched with CPU isolation applied to both the host and the guest, with osnoise running on top of it.
- Simultaneously, osnoise was launched on the host to trace the events on the CPUs where the virtual machine was executed.

By combining the existing osnoise tracepoint with the newly added one, the events responsible for causing spikes in the noise observed on the guest were identified. As seen previously on the guest, the results revealed that, even after system tuning, some housekeeping tasks were still executed on the isolated CPUs, introducing non-negligible noise to the virtual machine. This was due to the limitations of the experimental hardware, which featured only four available cores and could not completely isolate the CPUs. The experiment demonstrated that with these new features, osnoise not only quantifies virtualization-induced noise, but also provides insights into the events that trigger it, thereby expediting system debugging.



```
irq_noise:      local_timer:236 start 1077.013768122 duration 11004 ns
thread_noise:   qemu-system-x86: 3228 start 1077.013574395 duration 204793 ns
thread_noise:   rcuc/3:49 start 1077.013791159 duration 9895 ns
irq_noise:      irq_work:246 start 1077.013804872 duration 411 ns
thread_noise:   ksoftirqd/3:51 start 1077.013801996 duration 20362 ns
vm_noise:       vm_id:1 exit_cpu:3 exit_time:1077.013765798 entry_cpu:3 duration:62486 ns exit_reason:0x0001 overhead: 17998 ns
```

Figure 25: Osnoise tracer output showing an example of vmexits above 60 us caused primarily by host interferences.

6. Conclusions

As of today, the use of virtualization technologies in High Performance Computing and Real-Time applications has become an accepted practice. However, the standard tools and benchmarks used to assess the feasibility of the use of virtualization to run these kind of workloads lack a detailed explanation of the causes of performance deficits and the role of the hypervisor. This leaves to the user the effort to analyze the system to understand the origins of performance degradations. Therefore, this work proposes a shift in the way of debugging the systems where these workloads are executed. To do so, osnoise, a tool already available in the Linux kernel used to trace and measure the Operating System Noise of a system has been extended. Indeed, osnoise has as its main feature the ability to provide both the noise caused by the system and to point on where such noise has originated. Starting from this contribution, the tool has been extended to include in its output also the noise coming from the virtualization. As of now, osnoise combines the noises caused by KVM and QEMU. It would be a desired improvement to be able to distinguish the two, in order to have a better understanding of where the noise is coming from. In its base version, osnoise is merged with another tracer, timerlat [35]. Measures one of the main metrics used in real-time applications: scheduling latency. As in osnoise, it would be beneficial to extend it in order to account for the scheduling latency caused by the hypervisor and have one more tool to evaluate the performances of real-time applications when run inside a virtual machine.

References

[1] Kvm unit tests. https://www.linux-kvm.org/page/KVM-unit-tests.

[2] R. H. Arpaci A. C. Dusseau and D. E. Culler. Effective distributed scheduling of parallel workloads. *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1996.

[3] J. T. Brown A. J. Younge, R. Henschel and G. von Laszewski. Analysis of virtualization technologies for high performance computing environments. *IEEE International Conference*, pages 9–16, 7 2011.

[4] D. Laor U. Lublin A. Kivity, Y. Kamay and A. Liguori. kvm: the linux virtual machine monitor. *Proceedings of the Linux Symposium*, 1, 2007.

[5] R. W. Wisniewski F. J. Cazorla A. Morari, R. Gioiosa and M. Valero. A quantitative analysis of os noise. *IEEE International Parallel Distributed Processing Symp*, page 852–863, 2011.

[6] V. K. Koshelev A. O. Kudryavtsev and A. I. Avetisyan. Prospects for virtualization of high-performance x64 systems. *Programming and Computer Software*, 72:196–207, 2023.

[7] L. Abeni and D. Faggioli. An experimental analysis of the xen and kvm latencies. *IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 18–26, 5 2019.

[8] F. Bellard. Qemu, a fast and portable dynamic translator. *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 41, 2005.

[9] Shahid H. Bokhari. The linux operating system. *IEEE Computer Society Press*, 28:74–79, 1995.

[10] Paolo Bonzini. Realtime kvm. https://lwn.net/Articles/656807/.

[11] Felix Cloutier. Cpuid — cpu identification. https://www.felixcloutier.com/x86/cpuid.

[12] Glauber Costa. Kvm-specific msrs. https://kernel.org/doc/html/v5.9/virt/kvm/msr.html.

[13] D. Casini D. B. De Oliveira and T. Cucinotta. Operating system noise in the linux kernel. *IEEE Transactions on Computers*, 72:196–207, 2023.

[14] Daniel Casini D. B. De Oliveira, Tommaso Cucinotta and R. De Oliveira. Demystifying the real-time linux scheduling latency. *Leibniz International Proceedings in Informatics, LIPIcs*, 165(20), 2020.

[15] F. M. V. Ramos D. Kreutz and P. E. Verıssimo. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103:14–76, 2015.

[16] Mirosław Klinkowski Dariusz Wypiór and Igor Michalski. Open ran—radio access network evolution, benefits and market trends. *Appl. Sci. 2022*, 1, 2022.

[17] Felipe Oliveira David Beserra and Jean Araujo. Performance evaluation of hypervisors for hpc applications. *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 846–851, 2015.

[18] D. B. de Oliveira and R. S. de Oliveira. Timing analysis of the preempt rt linux kernel. *Softw., Pract. Exper*, 46:789–819, 2016.

[19] Kaivalya M. Dixit. The spec benchmarks. *Parallel Computing*, 17:1195–1209, 1991.

[20] DPDK Documentation. Poll mode driver. https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html.

[21] Jason Nieh Edouard Bugnion and Dan Tsafrir. x86-64: Cpu virtualization with vt-x. *Hardware and Software Support for Virtualization. Synthesis Lectures on Computer Architecture. Springer, Cham*, 2017.

[22] D. Kerbyson F. Petrini and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q,. *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 55–55, 2003.

[23] G. Massari F. Reghenzani and W. Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Computing Surveys*, 52:1–36, 2019.

[24] Felix Freitag Fernando Rodríguez-Haro and Leandro Navarro. A summary of virtualization techniques. *Procedia Technology*, 3:267–272, 2012.

[25] Center for Information Services and High Performance Computing (ZIH). Future grid. https://tu-dresden.de/zih/forschung/projekte/futuregrid?set_language=en.

[26] Linux Foundation. System management interrupts. https://wiki.linuxfoundation.org/realtime/documentation/howto/debugging/smi-latency/smi.

[27] Kumar S. Raj H. Schwan K. et al. Gavrilovska, A. High-performance hypervisor architectures: Virtualization in hpc systems. *HPCVirt*, pages 1–8, 2007.

[28] E. V. Hensbergen. The effect of virtualization on os interference. *IEEE Transactions on Computers*, 72:196–207, 2023.

[29] L. Abeni J. Lelli, C. Scordino and D. Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46:821–839, 2016.

[30] Jan Kiszka. Towards linux as a real-time hypervisor. https://static.lwn.net/lwn/images/conf/rtlws11/papers/proc/p18.pdf.

[31] Lawrence Livermore National Laboratory. Ftq/fwq. https://asc.llnl.gov/sites/asc/files/2020-06/FTQFTW_Summary_v1.1.pdf.

[32] Sasha Levin. hashtable: introduce a small and naive hashtable. https://lwn.net/Articles/510271/.

[33] L. De Simone M. Cinque, D. Cotroneo and S. Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 129:315–330, 2022.

[34] L. Mandyam and S. Hoenisch. Ran workload performance is equivalent on bare metal and vsphere. https://blogs.vmware.com/telco/ran-workload-performance-tests-on-vmware-vsphere/.

[35] D. B. De Oliveira. Timerlat tracer. https://docs.kernel.org/trace/timerlat-tracer.html.

[36] S. Boyd-Wickizer P. E. McKenney, J. Fernandes and J. Walpole. Rcu usage in the linux kernel: Eighteen years later. *SIGOPS Oper. Syst. Rev.*, 54:47–63, 2020.

[37] J. J. Dongarra P. Luszczek1 and D. Koester. Introduction to the hpc challenge benchmark suite. https://www.osti.gov/servlets/purl/860347.

[38] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architecture. *Communications of the ACM*, 17(7), 1974.

[39] The Linux Foundation Projects. Dpdk. https://www.dpdk.org/.

[40] IBM Research. rhype. https://github.com/machinaut/rhype.

[41] SDN and OpenFlow World Congress. Network functions virtualisation – introductory white paper – an introduction, benefits, enablers, challenges call for action. https://portal.etsi.org/NFV/NFV White Paper.pdf.

[42] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI'10. USA: USENIX Association*, pages 33–46, 2010.

[43] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. *Cluster Computing, 2004 IEEE*, 2004.

[44] Top500. Top 500 list: June 2023. https://www.top500.org/lists/top500/2023/06/.

[45] L. Tung. Spacex: We've launched 32,000 linux computers into space for starlink internet. https://www.zdnet.com/article/spacex weve-launched-32000-linux-computers-into-space-for-starlink-internet/.

[46] Rik van Riel. Real-time kvm from the ground up, linuxcon na 2016. https://wiki.linuxfoundation.org/_media/realtime/events/rt-summit2016/kvm_rik-van-riel.pdf.

[47] Frederic Weisbecker. Cpu-isolation by suse lab. https://www.suse.com/c/cpu-isolation-introduction-part-1/.

[48] J.J Wiley. Protection rings. *van Tilborg, H.C.A., Jajodia, S. (eds) Encyclopedia of Cryptography and Security*, 2011.

[49] Peter Xu. Oslat. https://github.com/xzpeter/oslat.

[50] Alexei Zakharov. Sysjitter. https://github.com/alexeiz/sysjitter.

Abstract in lingua italiana

Ad oggi, è prassi consolidata l'utilizzo delle tecnologie di virtualizzazione nelle applicazioni di High Performance Computing e Real-Time. La maggior parte delle applicazioni HPC si sta infatti spostando verso le infrastrutture offerte dal Cloud, grazie alla riduzione dei costi e alla flessibilità che esse comportano. Fanno eccezione le applicazioni sensibili alla sicurezza, come quelle del settore della Difesa, o le applicazioni con latenze nell'ordine dei nanosecondi, come quelle che si possono trovare, ad esempio, nel trading ad alta frequenza. Allo stesso tempo, l'uso di macchine virtuali ha guadagnato popolarità nei sistemi Real-Time per la facilità di integrazione e supporto nei sistemi a criticità mista. Infine, con l'avvento della tecnologia 5G, la virtualizzazione delle funzioni di networking sta guadagnando terreno, con servizi di rete a bassa latenza distribuiti su macchine virtuali. La comunità scientifica ha dimostrato come questi tipi di carichi di lavoro raggiungano prestazioni comparabili quando vengono eseguiti su una macchina virtuale e su bare-metal. Tuttavia, gli strumenti e i benchmark utilizzati per il debug dei sistemi in cui devono essere eseguite questo tipo di applicazioni non si concentrano sull'impatto che la virtualizzazione ha sulle prestazioni complessive e non forniscono le cause dei deterioramenti delle prestazioni che la virtualizzazione introduce. Questo lavoro evidenzia il ruolo della virtualizzazione nei deficit di prestazioni delle applicazioni a bassa latenza e migliora il debugging dei sistemi in cui vengono eseguiti questi carichi di lavoro. Lo fa estendendo Osnoise, un tracer del sistema operativo Linux, che consente di misurare e tracciare il rumore del sistema operativo, un problema ben noto nel settore HPC, con un impatto anche sulle applicazioni real-time. Osnoise fornisce sia le misure che l'identificazione delle sorgenti del rumore, in modo integrato, facilitando l'analisi e il debug del sistema. Partendo da questo contributo, questo lavoro estende osnoise seguendo un approccio simile, consentendo innanzitutto di misurare l'impatto che la virtualizzazione ha sul rumore complessivo del sistema operativo sperimentato da un carico di lavoro, e fornendo poi le informazioni per ricostruire gli eventi che portano al rumore di virtualizzazione misurato. Infine, viene mostrato come l'estensione sia utilizzabile per accelerare il debugging del sistema in cui è in esecuzione una macchina virtuale.

Parole chiave: KVM, Macchine Virtuali, Rumore del Sistema Operativo, Calcolo Ad Alte Prestazioni, Sistemi Real-Time

Acknowledgements

Per prima cosa, vorrei ringraziare il mio relatore, il Prof. Zaccaria, per i suoi consigli e per la disponibilità. Grazie per essere stato presente durante tutto il lavoro di tesi e per aver evidenziato pregi e difetti quando era necessario.
Un ringraziamento doveroso va a Paolo e Daniel; da voi ho imparato molto, siete stati fondamentali durante tutto lo svolgimento della tesi in cui mi avete seguito con costanza e dedizione, mostrando una disponibilità affatto scontata e per questo vi sono grato.
Un ringraziamento speciale va alla mia famiglia. Ai miei genitori, siete le mie colonne portanti, siete le prime persone su cui posso contare, so che ci sarete sempre per me ed io per voi. Mi avete supportato in ogni mia scelta e guidato quando ne avevo bisogno; devo a voi se sono la persona che sono.
A mia sorella Beatrice, per esserci sempre, per capirmi e cercarmi nei momenti in cui sono meno presente. So di poter contare su di te, in ogni momento, qualora ne avessi bisogno.
Un grazie ai miei amici di sempre: Luca, Alessandro ed Ettore. Anche se ci sentiamo di meno, il rapporto che ho con voi non è mai cambiato e sono sicuro che non cambierà.
Ai miei coinquilini, Kevin ed Annalisa, per aver creato un' atmosfera serena e felice in casa, non potevo chiedere di meglio.
Ad Ettore, Andrea, Giampiero e Riccardo, per aver reso questi anni a Milano fantastici, per tutte le esperienze condivise insieme e quelle che condivideremo.
Un grazie a Ledio, Edoardo, Annalisa, Stefania, Michela ed Andrea per tutte le serate e i momenti passati insieme in questi anni.
Un grazie a Denaldo, Kevin, Anne, Melissa, Samy ed Antonio, per aver reso i sei mesi a Barcellona uno dei periodi piu belli della mia vita e per essere ancora presenti.
A Pietro e Gregorio, per avermi sempre motivato e consigliato; devo anche a voi il percorso che ho deciso di intraprendere.
Infine un grazie a tutti i ragazzi di Perugia, per l'amicizia, nonostante le poche occasioni per stare insieme.