Paratick: Reducing Timer Overhead in Virtual Machines

Stijn Schildermans
Kris Aerts
stijn.schildermans@kuleuven.be
kris.aerts@kuleuven.be
KU Leuven Campus Diepenbeek
Diepenbeek, Belgium

Jianchen Shan
Hofstra University
New York, USA
Jianchen.Shan@hofstra.edu

Xiaoning Ding
New Jersey Institute of Technology
Newark, USA
xiaoning.ding@njit.edu

ABSTRACT

To this day, efficient timer management is a major challenge in virtualized environments. Contemporary timekeeping techniques in guest kernels frequently interact with timer hardware, which requires continual and costly hypervisor interference.

This paper proposes the concept of *virtual scheduler ticks*, which significantly reduces the need for guests to interact with timer hardware through the use of paravirtualization. Guests no longer program scheduler ticks, but rely on the host to inject its own ticks into vCPUs upon VM entry. We implemented virtual scheduler ticks in Linux/KVM under the name *paratick*.

We present a thorough performance analysis of paratick in the context of hardware-assisted X86 virtualization. Paratick reduces VM exits by up to 80%, enhancing system throughput by up to 125% and execution time by up to 15% for multithreaded applications relying heavily on blocking synchronization. For I/O-intensive applications, these numbers are respectively 45%, 30% and 25%.

KEYWORDS

timer, tick, virtualization, multithreading, I/O

ACM Reference Format:
Stijn Schildermans, Kris Aerts, Jianchen Shan, and Xiaoning Ding. 2021. Paratick: Reducing Timer Overhead in Virtual Machines. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472510

1 INTRODUCTION

Timekeeping is a fundamental duty of the operating system (OS). The OS assimilates hardware timekeeping devices and presents a unified timer API to applications [29]. Additionally, the OS keeps track of the passing of real time in the background, and performs general maintenance tasks such as scheduling, accounting, etc. at regular time intervals. In all popular contemporary general-purpose operating systems, all of these duties are driven by recurring OS-managed physical timer interrupts, called the *scheduler tick* [19].

Historically, the scheduler tick was programmed at a constant rate on all CPUs. Most modern operating systems use an optimized version of this mechanism, called *tickless kernel* operation [12]. In tickless systems, the scheduler tick is disabled whenever the OS deems it acceptable to do so. Most commonly, this is done whenever a CPU runs out of runnable tasks and enters an idle state.

In native environments tickless kernels are highly efficient, in contrast to classic periodic ticks. In virtualized environments -which are ubiquitous these days due to the advent of cloud computing- however, both classic periodic ticks and tickless systems may induce severe performance penalties. The former have been shown to cause unacceptable levels of overhead in heavily overcommitted virtualized systems because every vCPU -irrespective of its workload- requires frequent tick interrupt injection [34]. The latter suffer much less from this issue since idle vCPUs do not require tick interrupts. However, enabling/disabling the scheduler tick upon idle exit/entry is in itself a costly operation in a virtual environment, requiring hypervisor involvement. Certain workloads induce frequent brief idle periods by design, such as multithreaded applications employing blocking synchronization or I/O-heavy applications, thereby inducing massive virtualization overhead in tickless systems [32]. It is evident that a more intelligent solution is needed.

The main goal of this work is to drastically reduce virtualization overhead induced by scheduler tick management through paravirtualization. To this end, we introduce the concept of *virtual scheduler ticks*, which proposes to remove scheduler tick management entirely from guest kernels. Instead, the host utilizes the VM exits generated by its own tick interrupts to inject virtual ticks into each running vCPU. This may all but eliminate virtualization overhead induced by scheduler tick management.

We implemented and evaluated the concept of virtual scheduler ticks in Linux/KVM under the name *paratick*. Since our evaluation shows that paratick improves system throughput by up to 125% and application runtimes by up to 25% compared to the mainline Linux/KVM at the time of writing, we hope this work attracts the attention of operating system and hypervisor developers, especially since scheduler tick management in virtual machines has to our knowledge received very little attention in both literature and industry. Eventually, we plan to propose a patch for the mainline Linux kernel based on paratick.

1.1 Contributions

- We detail why neither classic periodic ticks nor tickless kernels perform satisfactorily in virtualized environments.

ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472510

- We introduce virtual scheduler ticks: a paravirtualization-based technique to drastically reduce virtualization overhead induced by scheduler tick management;
- We present and evaluate paratick; an implementation of virtual scheduler ticks in Linux/KVM.

2 BACKGROUND: TIMER MANAGEMENT

Many applications, as well as the OS itself, rely heavily on accurate time management. Because timer hardware is often complex and programming it may require expensive operations, many operating systems choose to implement a high level of abstraction in their timer APIs. Most often, application timers are managed as *soft interrupts*. This means that when an application programs a timer, often no actual timer hardware is programmed. Instead, the application timer is added to a dedicated data structure (e.g. the *timer wheel* in Linux [14]). Upon completion of any system call or hardware interrupt, the OS checks if any soft interrupts have expired and require servicing [30]. Therefore, timer management equates to managing the underlying mechanisms that invoke context switches and allow soft interrupts to be serviced.

Besides servicing soft interrupts, the OS must perform many other tasks, including resource accounting, scheduling, etc. at a regular interval. As mentioned in §1, for decades all mainstream operating systems have employed a periodic scheduler tick to achieve this [19]. Every CPU in the system is interrupted by a hardware timer (commonly the LAPIC timer in X86) at a regular interval, typically between one and ten milliseconds [34]. Upon receipt of the tick interrupt, the CPU passes control to the kernel, which then has the chance to perform the tasks described above.

While periodic scheduler ticks are simple and effective, they are not suitable for all environments. Particularly, it is not uncommon for modern SMP systems to have multiple idle CPUs for the majority of their lifespan. Because idle CPUs require negligible scheduling and bookkeeping work, processing scheduler ticks is a waste of resources for them. Especially for battery-powered systems such as smartphones this unnecessary resource usage can be problematic. Using classic periodic ticks, such systems may spend two thirds of their energy usage on processing scheduler ticks on idle cores [12]. Because of these issues, starting with Linux (2.6.21), the concept of *tickless kernels* was introduced, later to be adopted by most other mainstream operating systems [23]. Such systems can be viewed as systems based on a regular periodic scheduler tick, with additional code that identifies scenarios in which the tick is not useful. If such a scenario is detected, the tick is deferred or disabled entirely.

Generally, tickless kernels disable the scheduler tick whenever a CPU runs out of runnable tasks and enters an idle state. Figure 1 schematically shows this algorithm in detail, as implemented in Linux. Though implementation details may differ for other operating systems, the general algorithm is similar.

Handling tick interrupts in tickless kernel mode is largely identical to the tick handling process for classic periodic ticks, as shown in fig. 1a. Whenever a tick interrupt is received, the kernel performs the necessary tasks, reprograms the tick timer to expire after a fixed amount of time, and returns. The only difference between the tickless tick handler and the classic one is that the former checks whether the tick has been deferred or disabled by the time the tick interrupt handler was invoked. This may happen in exceptional circumstances. If so, the reprogramming step is skipped.

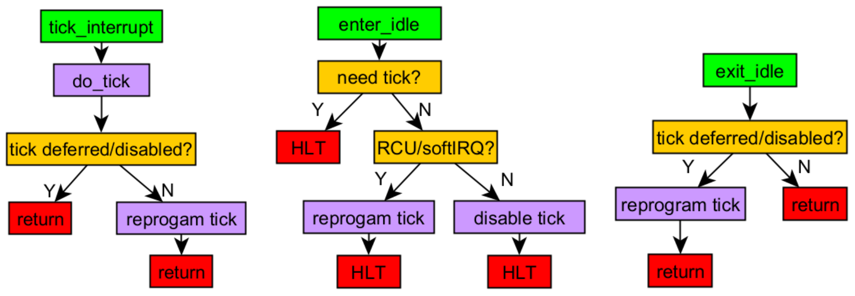
Figures 1b and 1c represent the core of the tickless code in Linux. Whenever a CPU is about to enter the idle loop, the code checks if any system component (RCU, irq work,…) explicitly needs the tick to remain enabled or if the next RCU event or soft interrupt falls within the next tick period. If so, the tick is not disabled and the CPU immediately enters the idle loop, as shown in fig. 1b. If neither of these conditions are met, the algorithm finds the next scheduled RCU callback or soft interrupt. The tick timer is then reprogrammed to expire at the expiry time of that event. If there are none, the tick is disabled entirely. Upon exiting the idle state, the algorithm checks if the tick has been deferred or disabled upon idle entry. If so, it is reprogrammed to expire at the regular tick interval, as shown in fig. 1c.

This work only focuses on classic periodic ticks and tickless kernels as shown in figure 1. After all, to our knowledge, these are the only two mechanisms commonly used by operating systems today. Besides these operating modes, called resp. *periodic* and *dynticks idle* mode, another mode of operation exists with regard to the scheduler tick, namely *full dynticks* mode. This mode disables the tick on CPUs that have at most one runnable task. This may be beneficial for systems that have predictable workloads that do not spawn more tasks than the number of CPUs in the system. Because this mode of operation targets highly specific workloads and to our knowledge only exists in Linux as an option for advanced users, we refrain from elaborating further on it.

3 VIRTUALIZING THE SCHEDULER TICK

Due to the advent of cloud computing ever more applications are running in a virtual environment. Modern virtualization techniques are mostly effective at mitigating virtualization overhead. However, effectively virtualizing the scheduler tick is one of the last remaining challenges [32]. In this section, we describe the state of the art regarding timer virtualization in the context of Linux running on an X86-platform employing hardware-assisted virtualization. Although other systems may use different mechanisms, the same general principles apply [5, 29].

If available, Linux uses the per-CPU *time stamp counter (TSC)*, which is the most accurate timer hardware available for programming timers [2]. It is armed by writing the desired expiration time to the `TSC_DEADLINE` MSR. When the TSC value reaches said expiration time, the LAPIC generates a *local timer interrupt*. In native environments, this process has a very low cost. In virtualized environments however, each write to the `TSC_DEADLINE` MSR must be intercepted by the hypervisor, as its current value may correspond to a timer from the host or another VM. Moreover, the interrupt generated as the timer expires generates another VM exit, as the hypervisor must determine if the interrupt belongs to the currently running VM, the host or another VM. Some hypervisors (e.g. KVM) optimize this process by using the preemption timer rather than the LAPIC timer to signal guest timer interrupts. Upon each VM exit induced by a guest attempting to write to the `TSC_DEADLINE` MSR, the hypervisor arms the preemption timer for the vCPU in question, but leaves the `TSC_DEADLINE` MSR untouched. When the



(a) Physical tick handler (b) Idle entry (c) Idle exit

Figure 1: Schematic representation of standard tickless kernel operation in Linux.

preemption timer expires, a (less costly) VM exit is triggered which allows the hypervisor to inject a timer interrupt [1].

From the above, it is clear that handling scheduler ticks is a costly process in virtualized environments. The magnitude and nature of this cost may however vary greatly depending on the workload and whether the system is employing classic periodic ticks or a tickless kernel. Below we describe each of these scenarios in detail.

3.1 Classic Periodic Tick

As stated in §2, systems employing a classic periodic scheduler tick program a tick interrupt at a fixed rate on every (v)CPU, irrespective of the workload. Thus, based on the above, a system hosting a number of VMs n_{VM} using classic periodic ticks, each having a number of vCPUs n_{vCPU} and a tick frequency f_{tick}, will always incur the following number of VM exits related to timer management over a time period t:

$$exits = 2 \times t \times \sum_{n=1}^{n_{VM}} (n_{vCPU} \times f_{tick})$$

Especially for heavily overcommitted systems -where each physical CPU is time-shared between multiple vCPUs- the host may spend exorbitant resources on processing scheduler ticks. Namely, the running vCPU is suspended whenever a tick interrupt arrives for a descheduled vCPU, even if the latter is idle. Since one of the main applications of virtualization is consolidating workloads, such overcommitted scenarios where the majority of vCPUs are idle for the majority of the time are not rare. While the virtualization overhead for each individual vCPU is limited, system throughput may be severely reduced [34].

3.2 Tickless Kernels

Tickless kernels are often depicted as almost purely beneficial [12, 34]. While in native environments this claim may hold true, in virtualized environments their benefits are less clear. While tickless kernels do reduce the number of timer interrupts generated by the VM, which especially on overcommitted systems improves system throughput significantly, they must reprogram the tick timer upon each idle entry/exit, as described in §2. Since this reprogramming requires a write to the `TSC_DEADLINE` MSR and thus induces a VM exit, the number of VM exits induced by tick management in a tickless system can be described as follows:

$$exits = 2 \times t \times \sum_{n=1}^{n_{VM}} \left(L_n \times n_{vCPU} \times f_{tick} + \frac{(1 - L_n) \times n_{vCPU}}{T_{idle}} \right)$$

With L_n the VM load expressed as a ratio of the utilized and maximum VM throughput and T_{idle} the average idle period during the time t. Thus, the term $L_n \times n_{vCPU} \times f_{tick}$ represents active vCPU operation and the term $\frac{(1-L_n)\times n_{vCPU}}{T_{idle}}$ represents the number of transitions between active and idle states during the time t.

From the above, it is evident that for tickless kernels to be efficient in virtualized environments, the average idle period T_{idle} must be long relative to the total CPU time spent on idling ($t \times (1 - L_n) \times n_{vCPU}$), thus minimizing the number of transitions between idle and active vCPU states. While this holds true in most cases, certain workloads, such as multithreaded applications employing blocking synchronization and I/O-heavy tasks, may exhibit the opposite behavior. Regarding the former, critical sections are often no longer than a few microseconds. Therefore, synchronizing threads may block and unblock thousands of times per second. Previous work has shown that for such workloads, systems hosting tickless guests may spend 15% of their CPU time on processing VM exits related to tick management [32]. Regarding the latter, while I/O latencies vary greatly between devices, most are no more than a few ms. Given that most applications block on each I/O transaction until it is completed [35], great numbers of transitions between idle and active states may occur for I/O-heavy applications. In particular for high-performance I/O devices, this may induce severe virtualization overhead. Thus, tickless kernels are certainly not a silver bullet in all circumstances.

3.3 To Tick or not to Tick?

The above indicates that both periodic ticks and tickless kernels may induce severe performance issues in a virtualized environment. However, how they exactly relate to each other and which -if any- is to be considered superior is still unclear. Below we illustrate this by studying several concrete hypothetical virtualized workloads:

- **W1**: an idle VM with 16 vCPUs;
- **W2**: 4 idle VMs with 16 vCPUs each;

	W1	W2	W3	W4
periodic ticks	40 000	160 000	40 000	160 000
tickless	0	0	60 000	240 000

Table 1: Number of VM exits induced by periodic ticks and tickless kernels in a variety of scenarios.

- **W3**: a workload using 16 threads, synchronizing 1000 times per second through blocking synchronization, executed in a single VM with 16 vCPUs;
- **W4**: 4 concurrent copies of W3, each in a VM with 16 vCPUs.

Table 1 shows the amount of VM exits related to scheduler tick management induced by each of the above workloads when all of the VMs use resp. periodic ticks or tickless kernels with a tick frequency of 250 Hz, assuming the workloads are run for 10 seconds on a system with 16 physical CPUs. All values are calculated based on the formulas derived in §3.1 and §3.2.

Table 1 shows that for low-intensity workloads where the system is mostly idle, tickless kernels are vastly superior to periodic ticks. However, for high-intensity workloads which frequently switch between idle and active states, periodic ticks gain the upper hand. Specifically, tickless kernels are preferable as long as the average idle period T_{idle} is longer than the average vCPU tick period divided by the number of vCPUs sharing the same physical CPU. With tick periods commonly ranging between 1 and 10 ms, it is evident that for many workloads, this is not the case. Given that parallel computing has become the norm these days and more efficient I/O devices continue to emerge (e.g., datacenter network, NVMe storage), demand for better handling of microsecond-level idle periods continues to rise [8]. Stimulated by workloads such as AI and blockchain, various highly parallel accelerators (e.g., GPGPU and TPU) are being designed and deployed. Fine-grained computation offloads to such accelerators incur similar idle periods. Thus, neither classic periodic ticks nor tickless kernels exhibit acceptable levels of virtualization overhead for all common workloads. It is clear that an alternative method for managing scheduler ticks in virtualized environments is needed.

4 VIRTUAL SCHEDULER TICKS

To combat the issues highlighted in §3, we propose to reconsider the concept of scheduler ticks in virtualized environments. After all, as established in §2, scheduler tick management is a fundamental duty of the OS. In a virtualized environment, the hypervisor acts as the OS with regard to hardware management, essentially taking over this duty from the guest kernels. Since the scheduler tick is an OS-level mechanism designed solely for the purpose of tying system time to physical time by interacting with timer hardware, we argue that in a virtualized environment, management of the scheduler tick should be the sole responsibility of the hypervisor. Thus, a guest should be able to request scheduler ticks from the hypervisor much like applications may request system services from the OS. The hypervisor is responsible for performing the necessary hardware interactions to provide this service. This is the basic idea behind the concept of *virtual scheduler ticks*. Below we elaborate on the design of this concept and establish concrete design goals.

4.1 Design

From a technical perspective, the concept of virtual scheduler ticks may be interpreted as paravirtualizing the scheduler tick. The guest kernel must be modified so that it no longer programs its own scheduler tick. This obviously also eliminates the need for the VM to enable/disable the scheduler tick upon idle exit/entry. Instead, vCPUs rely on the hypervisor to inject virtual scheduler ticks. This can easily be achieved, since irrespective of the VMs it is hosting, the hypervisor must implement its own periodic interrupt to perform its basic functions, such as scheduling. These host scheduler ticks must interrupt any running vCPUs and pass control to the hypervisor. When vCPU execution is resumed, the hypervisor may inject a virtual tick interrupt, which the VM may handle as if it were its own physical tick interrupt.

Because the above method relies on each vCPU being interrupted by the host scheduler tick (and thus actively running), it does not suffice when the guest is idle or the physical CPU is time-shared with other vCPUs or host tasks. Therefore, the time of the last virtual tick injection must be accounted for each vCPU. On each VM entry, the host checks if the last virtual tick injection predates the requested tick interval for that vCPU. If so, a virtual tick is injected and the current time is recorded as the last tick. Furthermore, to ensure that idle vCPUs are woken up by the hypervisor in a timely manner despite not receiving any virtual ticks after they have been descheduled, upon idle entry, the guest checks if there are any soft interrupts or RCU tasks scheduled. If so, a timer is programmed to expire in accordance with the closest of these events. We heuristically decide not to disable this timer upon exiting the idle state, as the overhead induced by a single timer is negligible and it is likely that the vCPU will re-enter an idle state before the timer has expired. If the timer were to be disabled upon idle exit, the timer would need to be reprogrammed upon idle entry, thus inducing 2 VM exits. This mechanism is comparable to that employed by tickless kernels, as shown in figure 1b.

The above assumes that the host tick frequency equals that of the guests. Since this cannot be guaranteed, the guest should declare its tick frequency to the host during the boot sequence through a hypercall. If the host tick frequency is a multiple of that of the guest, no further actions are needed. If not, the host should program the guest preemption timer such that virtual ticks may be injected at the correct rate. Note that this does not introduce meaningful overhead, since if the guest were to program its own tick interrupts, two VM exits would be generated each tick period as well.

4.2 Performance Implications

The virtual scheduler tick concept drastically reduces the number of VM exits required for guest-level scheduler tick management. It thus eliminates the issues described in §3. While the guest may still induce some VM exits when it needs to program a timer upon idle entry, this number is negligible compared to the numbers induced by periodic ticks and tickless kernels for almost any workload. In fact, virtual scheduler ticks is guaranteed to never induce more timer-related VM exits than tickless kernels, as the latter require the timer hardware to be touched on practically every idle entry/exit.

Compared to classic periodic ticks, virtual scheduler ticks conceptually offer a tangible performance improvement, in particular when guests are mostly idle and/or the host is overcommitted. Compared to tickless kernels on the other hand, the benefits are dependent on the workload. In light use cases, tickless systems induce very little virtualization overhead to begin with, as shown in §3.3. Therefore, a significant reduction in tick-related virtualization overhead may still only improve performance marginally. For multithreaded workloads however, system throughput may be improved drastically. Nevertheless, application execution times may not improve accordingly because the execution time of multithreaded applications is determined solely by the critical path [38]. Therefore, only VM exits incurred upon idle exit (idle entry is by definition not part of the critical path) and belonging to a single execution path influence application execution time. Thus, for multithreaded workloads, a significant improvement in system throughput is expected, which may however translate to a much smaller improvement in application execution time. Regarding I/O-heavy workloads, application execution time is dominated by waiting for the I/O device. Additionally, I/O is known to generate many VM exits not related to tick management [17]. Thus, for these workloads, a significant improvement in system throughput is expected for low latency I/O devices such as SSDs and high-performance NICs, since for such devices the time spent waiting for the device is limited relative to the time spent on processing the I/O. For high latency I/O devices such as HDDs on the other hand, the potential for improvement is limited. In any case, said throughput improvement will likely lead to a comparable improvement in application execution time, since almost all VM exits incurred upon idle exit are likely part of the critical path, since when an I/O interrupt arrives, any delay directly delays the next I/O operation.

5 PARATICK

We implemented virtual scheduler ticks in Linux/KVM under the name *paratick*. Below we describe this implementation, using Linux 5.10.26 as the baseline. The source code is freely available[1].

5.1 Host

Implementing paratick requires minimal effort on the host side. Firstly, a field was added to the struct KVM uses to represent a vCPU internally (`kvm_vcpu`) representing the time of the last virtual tick injection. Secondly, we modified the main KVM loop which is responsible for running vCPUs. If the vCPU has a pending local timer interrupt upon VM entry, the `last_tick` field of the `kvm_vcpu` struct is updated. We thus assume that the local timer interrupt to be injected will act as a tick interrupt. While this can not strictly be guaranteed, we heuristically assume that the interrupt was likely programmed by the guest-side paratick code upon idle entry. Moreover, upon receipt of any interrupt Linux by default performs basic timekeeping work [4]. After extensive testing, we determined that this heuristical optimization is valid. If no local timer interrupt is to be injected upon VM entry, paratick evaluates if the time elapsed since the last tick injection is greater than the tick period. If so, a virtual tick interrupt is injected and the `last_tick` field of the `kvm_vcpu` struct is updated. We reserve vector 235 for this purpose. Fig. 2 illustrates the above schematically.

[1]https://github.com/StijnSchildermans/paratick.git

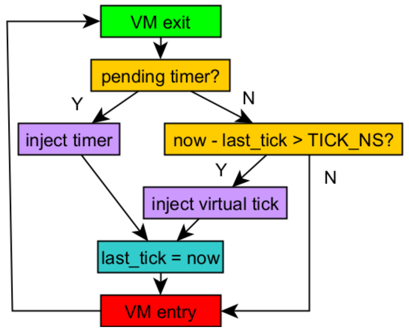


Figure 2: Schematic overview of host-side paratick code.

For our purposes, the above host-side implementation suffices since both the host and guest run the same version of Linux, with the same tick frequency. Therefore the guest will certainly receive ticks at the desired frequency. However, if we can not guarantee that the guest and host kernels are using identical tick frequencies, additional code must be added in accordance with the design described in §4.1. We leave the implementation of this feature for future work, as it is does not add any value in terms of assessing paratick from a research perspective.

5.2 Guest

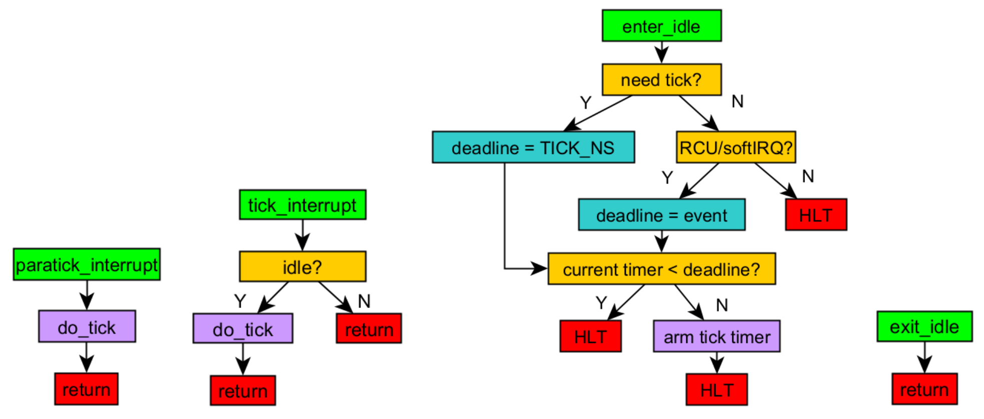
The guest-side implementation of paratick is somewhat more pervasive than that on the host side. Still, all the changes can be implemented by altering just the main scheduler tick source file (`kernel/time/tick-sched.c`). Figure 3 schematically shows the high-level guest-side paratick implementation, arranged in such a way that it can easily be compared to the regular tickless Linux kernel, as shown in figure 1.

Figure 3 shows that paratick preserves the basic structure of the tickless Linux kernel, while adding an extra handler for virtual tick interrupts. Below, we describe all the changes made in detail.

5.2.1 System Boot. High-resolution timers, upon which both tickless and paratick mode rely, only become available partway through the boot process. Before this time, the system uses a regular periodic scheduler tick. Therefore, we integrate the paratick initialization code, which encompasses installing an interrupt descriptor for the virtual scheduler tick interrupt vector, with the tickless initialization code. The periodic scheduler tick is disabled as soon as the switch to paratick mode is made. Any virtual ticks arriving before the switch to paratick mode are rejected.

5.2.2 Virtual Tick Handling. As figure 3a shows, a handler for the virtual scheduler tick was added. This handler performs the same functions as the standard Linux tick handler (fig. 1a), with the exception that it never (re)arms a physical timer before returning.

5.2.3 Physical Tick Handling. As described in §4.1, paratick may require a physical timer to be programmed upon idle entry. Figure 3b shows that the handler for this timer first checks if the vCPU is still idle when receiving the interrupt. If so, this interrupt is likely crucial to the system and is treated as a virtual tick interrupt. If not, the vCPU is currently operating normally, meaning virtual ticks



(a) Virt. tick handler **(b) Physical tick handler** **(c) Idle entry** **(d) Idle exit**

Figure 3: Schematic representation of guest-side paratick code.

are actively being injected. There is thus no need to perform any tick-related work and the handler returns.

5.2.4 Idle Entry. To determine whether a physical timer should be programmed upon idle entry, paratick can largely recycle the idle entry code from the vanilla kernel, as is evident by comparing figures 3c and 1b, with the important distinction that the status quo for paratick is that no tick is programmed and the idle entry code must check whether a timer should be set, while the status quo for tickless operation is that a timer is set and the idle code should determine whether to disable the tick. Concretely, this means that if the recycled tickless idle entry code determines the tick must be retained, paratick programs a timer to expire at the regular tick interval. Otherwise, paratick checks if a timer must be set to make sure the vCPU is woken up at expiry time of the next RCU event or soft interrupt, again recycling existing tickless kernel code. If so, the determined deadline is compared to the current expiry time of the tick timer, since as described in §4.1, the timer may not yet have expired after having been set at a previous idle entry. Only if the timer is not running or the newly determined expiry time is sooner than that the timer is currently using, it is (re)programmed.

5.2.5 Idle Exit. Because as described in §4.1 we heuristically determined that it is beneficial not to disable any physical timers set at idle entry upon idle exit, no action must be taken when a vCPU returns from idle, as shown by figure 3d. This stands in contrast to the tickless kernel implementation in Linux, which must re-enable the tick timer at (almost) each idle exit (see figure 1c).

6 EVALUATION

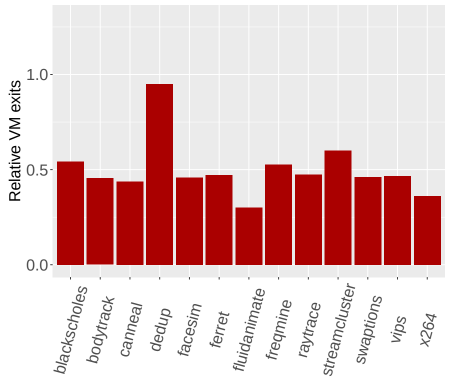
In this section we evaluate the effectiveness of paratick by comparing it to the standard Linux kernel. We start by evaluating the performance of computation-intensive sequential workloads -which we do not expect to benefit from paratick since they do not induce frequent transitions between idle and active CPU states- to assess if paratick itself introduces significant performance overhead. Afterwards, we assess multithreaded and I/O-intensive workloads, which are the main targets of this work, as established in §4.2.

The test system is a NUMA server with 4 sockets, each featuring 20 CPUs and 64 GB of RAM. Linux/KVM was installed on this machine, using kernel 5.10.26. We disabled pause loop exiting (PLE) because this optimization is only beneficial in overcommitted environments, where a physical CPU is time-shared between vCPUs. When this is not the case, any VM exits triggered by PLE unnecessarily degrade performance, possibly distorting test results [32]. Additionally, we disabled halt polling because it may consume large amounts of CPU cycles in an effort to slightly improve execution times. In some cases, a more efficient execution may lead to seemingly worse performance because faster execution increases thread contention, which will lead to increased halt polling cycles without improving execution time tangibly [32].

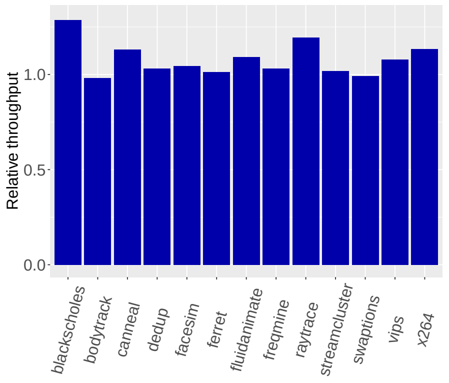
All VMs use Ubuntu 20.04 as the OS, running Linux 5.10.26 in the default 'dynticks idle' mode. Since kernels using classic periodic ticks are rare nowadays and we already compared them to tickless kernels in §3.3, we omit directly comparing paratick to classic periodic ticks. Readers may nevertheless infer such a comparison from combining the results in this section with those in §3.3.

For all experiments, we assess three metrics:

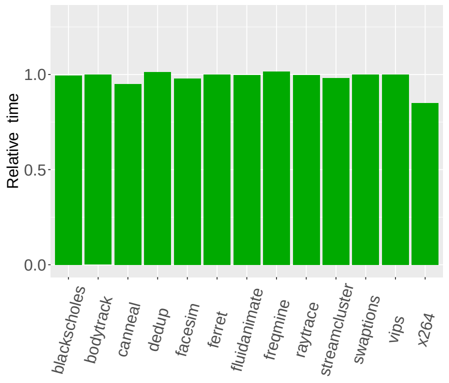
- **VM exits**: VM exits are the main source of host-level hardware assisted virtualization overhead [32]. Since paratick aims to eliminate the majority of writes to the `TSC_DEADLINE` MSR and associated VM exits, they show to what extent the basic goal of paratick has been achieved.
- **System throughput**: System throughput shows the effect of paratick on system resources. Since this metric takes all resources spent into account -useful work and overhead alike- it places the virtualization overhead reduction paratick yields into perspective.
- **Execution time**: Especially for multithreaded applications, there is no strong correlation between system throughput and application performance, since system resources spent on any execution path other than the critical path do not alter execution time [32]. Thus, application execution time



(a) VM exits



(b) System throughput



(c) Execution time

Figure 4: Relative performance of paratick compared to vanilla Linux for sequential PARSEC workloads.

is measured independently to provide an accurate depiction of system performance improvement visible to end users.

All experiments described in this section were repeated until their results stabilized. The displayed results are therefore the average of 3 to 15 iterations. Despite our best efforts however, a deviation of 5% is possible due to the multitude of non-deterministic factors to be taken into account (e.g. scheduling deviations).

6.1 Sequential Workloads

As described in §4.2, paratick is not expected to benefit low-intensity, I/O-lean workloads. Conversely, any overhead introduced by paratick itself would still be present. Because of this, sequential, computation-intensive workloads allow us to estimate the gross cost of paratick, irrespective of potential performance gains.

For this first set of experiments, the PARSEC benchmark suite was run in sequential mode on a VM with 1 vCPU. This benchmark suite contains 13 varied, realistic computation-intensive workloads [10]. Although PARSEC mainly targets parallel systems, all workloads can be executed sequentially as well. VM exits and application execution time may be measured directly using `perf`[2]. We use CPU cycles as a measure for system throughput. Although the latter is determined by many other factors, sections 4 and 5 make it clear that CPU cycles is the dominant metric being improved by paratick (through eliminating VM exits). Therefore, the improvement in CPU usage represents the maximum throughput improvement paratick may achieve. Figure 4 shows the results for each benchmark individually. Because from this figure alone it is unclear what the overall performance benefit of paratick is for the studied workloads due to the variance between individual benchmarks, table 2 shows the aggregated results for all benchmarks.

Figure 4a shows that even for low-intensity workloads, paratick reduces the number of VM exits drastically. Indeed, for such workloads, very few VM exits are induced in tickless kernel mode, as shown in §3.3. A large portion of these few VM exits are caused by 3 operations: arming the guest tick timer, delivering host ticks and delivering guest ticks. Since paratick eliminates 2 of these 3 major causes of VM exits, it has a highly positive effect on virtualization overhead for low-intensity workloads.

VM exits	System throughput	Execution time
-50%	+7%	-2%

Table 2: Average performance improvement of paratick accross all PARSEC benchmarks in sequential mode.

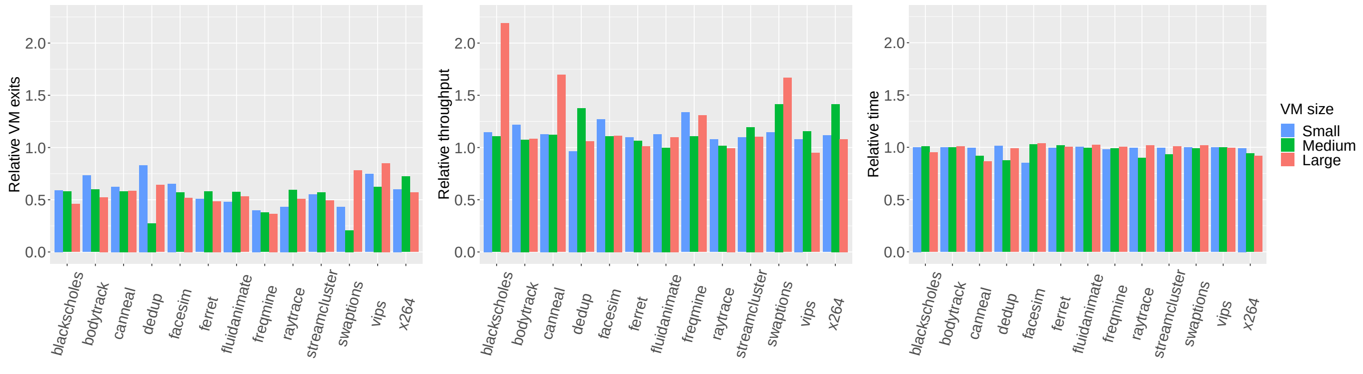
Despite figure 4a showing excellent results, figures 4b and 4c indicate that paratick only marginally improves system throughput and application performance for low-intensity workloads. This is in line with the expectations laid out in §4.2. Thus, even though the number of VM exits is reduced drastically, the amount of resources spent processing them is negligible relative to those spent on the workload itself. More importantly, these figures show that even in scenarios where paratick offers negligible benefits, workload latency and system throughput are not affected negatively.

6.2 Multithreaded Workloads

Having established that paratick does not introduce tangible gross overhead, we move on to assessing its potential benefits by evaluating its performance for workloads that conceptually benefit the most, being computation-intensive multithreaded applications (see §4.2). To cover a broad range of real-world environments, we create three test scenarios: a 'small' VM with 4 vCPUs collocated on the same NUMA socket, a 'medium' VM with 16 vCPUs spread over 2 NUMA sockets, and a 'large' VM with 64 vCPUs spread over 4 sockets. In each of these scenarios, we execute the PARSEC benchmark suite with the level of parallelism equal to the number of vCPUs in each scenario. All metrics are measured as in §6.1. Equally analogously to §6.1, figure 5 displays the results for all individual benchmarks and table 3 shows the aggregate results of all the benchmarks in each test scenario.

Figure 5a shows that for multithreaded workloads, paratick reduces the relative number of VM exits compared to tickless kernel operation by roughly the same amount as for sequential workloads. Nevertheless, figure 5b indicates that for several of these workloads

[2] https://man7.org/linux/man-pages/man1/perf-record.1.html

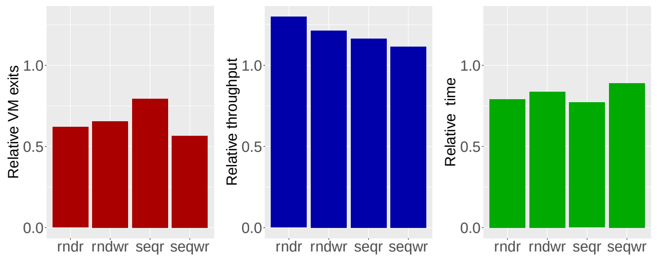


(a) VM exits (b) System throughput (c) Execution time

Figure 5: Relative performance of paratick compared to vanilla Linux for multithreaded PARSEC workloads.

VM size	VM exits	System throughput	Execution time
Small	-42%	+12%	-1%
Medium	-47%	+13%	-3%
Large	-44%	+16%	-1%

Table 3: Average performance improvement of paratick accross all PARSEC benchmarks in all tested scenarios.



(a) VM exits (b) System throughput (c) Execution time

Figure 6: Relative performance of paratick compared to vanilla Linux for I/O-intensive workloads.

-in contrast to sequential ones- paratick drastically improves system throughput. This is not illogical, since multithreaded workloads are known to induce many more VM exits than their sequential counterparts [32]. This means that the same relative reduction in VM exits translates to a much greater relative performance improvement for multithreaded workloads than for sequential ones.

The results in figure 5b show a large inter-benchmark variance. This is to be expected, since as outlined in §4.2, paratick specifically reduces blocking synchronization cost. Not all multithreaded workloads rely on this mechanism to the same extent. Blocking synchronization also lies at the heart of table 3 indicating increased paratick effectiveness as VM size grows. Namely, the level of parallelism dictates the amount of thread contention and therefore the amount of switches between running and blocked states.

Figure 5c confirms that as described in §4.2, the large throughput gain shown in figure 5b does not translate to a comparable reduction in application execution times, implying that the VM exits eliminated by paratick are mostly not part of the critical path. Nevertheless, improved throughput in itself is highly beneficial, since it reduces energy consumption and allows the host to perform more tasks in a given amount of time. Moreover, in scenarios where system resources are saturated, for example when runnable vCPUs from other VMs are waiting in the run queue, throughput improvement may directly improve application performance, since in such scenarios resource availability dictates the execution time of the critical path and thus of the entire application.

6.3 I/O-Intensive Workloads

The final class of workloads described in §4.2 as conceptually benefiting from paratick is that of I/O-intensive applications. To assess the veracity of this claim, the *fio* benchmark from the Phoronix benchmark suite [3] was executed in a VM with 1 vCPU. Sequential read (seqr), sequential write (seqwr), random read (rndr) and random write (rndwr) performance were independently evaluated. Each of these categories contain aggregated results for block sizes varying between 4kB and 256kB. For all tests, we opted for the sync I/O driver as synchronous I/O is much more popular than its asynchronous counterpart, due to the complexity and limited flexibility of the latter [35]. Direct I/O was disabled as is common practice. Buffering I/O was disabled to simulate reading/writing large data sets. Contrary to the PARSEC benchmark suite, phoronix-fio allows direct measurement of I/O throughput. Since I/O operations are the sole system bottleneck, I/O throughput equates to system throughput for this use case. In a more generalized setting, this represents the maximum throughput improvement paratick may achieve for I/O-intensive applications. The other metrics were measured in the same way as in §6.1 and §6.2. Again analogously to §6.1 and §6.2, figure 6 shows the results for each category individually while table 4 shows the aggregated results of all categories.

VM exits	System throughput	Execution time
-34%	+20%	-18%

Table 4: Average performance improvement of paratick across all tested phoronix-fio benchmarks.

Figure 6a indicates that also for I/O-intensive workloads, paratick significantly reduces virtualization overhead. The reduction in VM exits is however somewhat smaller than for the application classes discussed above. This is to be expected, because I/O is notorious for inducing high virtualization overhead in general [17] and the test system does not possess a high-end SSD device supporting single root I/O virtualization (SR-IOV) [16]. Therefore, timer-related VM exits make up a relatively small part of the total number of VM exits such workloads induce.

Figure 6b shows that paratick may yield a significant throughput improvement for I/O-intensive applications. Interestingly, the average throughput improvement displayed in table 4 is not much lower than the average reduction in virtualization overhead. This indicates that virtualization overhead makes up a significant part of the total system resources utilized by I/O-intensive applications.

Lastly, figure 6c and table 4 reveal that for I/O-intensive applications, throughput improvement translates almost directly to improved application execution times. This makes sense, since as described in §4.2, at least half of the VM exits eliminated by paratick are part of the critical path for single-threaded I/O-intensive applications. Moreover, figure 6c indicates that read operations benefit the most from paratick. Given that read latencies are lower than write latencies and reads are mostly synchronous while writes are generally asynchronous, more switches between active and idle vCPU states are performed when reading data as opposed to writing it in the same amount of time. Therefore, the VM exits eliminated by paratick make up a larger percentage of the total application runtime. This also indicates that paratick's performance benefits will only increase as time goes on, since state-of-the-art storage devices supporting SR-IOV sport much lower access latencies.

7 RELATED WORK

Timer overhead in virtualized environments has received little attention in literature. Only a few papers [11, 18, 37] target timekeeping in VMs and its effects on scheduling and application performance [7, 20, 22]. One major reason for this is that most recent efforts regarding reducing virtualization overhead focused on more dominant forms thereof [16], including the lock-holder preemption problem [13, 21, 33], blocked-waiter wakeup problem [15], lock-waiter preemption problem [27], TLB shootdown preemption problem [28, 31], etc. All of these problems share the same fundamental cause, namely vCPU discontinuity [6, 13, 25]. However, recent improvements to virtualization technology have largely mitigated these issues [9, 17, 24, 39]. This makes optimizing timer management one of the last significant remaining challenges regarding efficient virtualization of the X86 platform [32].

Although the problem of scheduler tick management in virtualized environments has to our knowledge never been addressed explicitly in literature, some studies indirectly offer potential solutions. *OSv* [26], a new guest operating system designed specifically for cloud computing employs a fully tickless design, utilizing a high resolution clock for time accounting as long as the use case only calls for a single application to be run at a time. While for many cloud applications such a design suffices, it is obviously not applicable to general-purpose operating systems. A more promising existing solution in that regard is *direct interrupt delivery (DID)* [36]. DID directly delivers timer interrupts to the target VM, bypassing VM exits through clearing the external interrupt exiting (EIE) bit in VMCS (Virtual Machine Control Structure). In addition, it programs the hardware not to perform VM exits upon writes to timer-related MSRs. This solution does however not come without cost, since timers set by the hypervisor and descheduled vCPUs are restricted to a designated core, which can become a bottleneck under heavy loads. Moreover, the designated core can not be used by VMs. This can be interpreted as a static virtualization overhead inversely proportional to the number of CPUs in the system. For all but the most high-end contemporary systems, this results in a non-negligible loss in system throughput.

From the above, we conclude that to our knowledge, virtual scheduler ticks is the only low-overhead solution to the problem of excessive timer-related virtualization overhead applicable to general-purpose operating systems and on limited as well as state-of-the-art hardware platforms.

8 CONCLUSION

Even in state-of-the-art virtualized environments, timer management remains a major source of virtualization overhead. In the context of hardware-assisted virtualization of the X86 platform, this overhead manifests itself as VM exits induced by managing the scheduler tick. In this paper, we have shown that classic periodic ticks as well as tickless kernels may induce excessive amounts of said VM exits in specific scenarios. Since to our knowledge, all general-purpose operating systems rely on either of these mechanisms, addressing this issue is paramount.

This paper introduced the concept of virtual scheduler ticks in an effort to address the above problem through the use of paravirtualization. We have shown the potential of this concept by implementing it in Linux/KVM under the name 'paratick' and demonstrating that it may enhance system throughput greatly by drastically reducing the number of VM exits related to scheduler tick management. Especially multithreaded applications relying heavily on blocking synchronization and I/O-intensive applications benefit. For the former, the system throughput improvement yielded by paratick translates to only a minor application execution time reduction, since many of the VM exits eliminated by paratick are not part of the critical path. For the latter however, performance gains are in accordance with system throughput amelioration.

To our knowledge, we are the first to address excessive tick-induced virtualization overhead directly in a generally applicable way. The only drawback of virtual scheduler ticks that we were able to identify during the course of the project this paper covers is that it relies on paravirtualization and therefore requires modifications to the guest kernel. This complicates dissemination, especially towards closed-source systems. Whenever this drawback is not a concern however, paratick is a clear improvement over regular tickless kernels as well as classic periodic ticks in virtualized environments.

Therefore, as future work, we aim to further refine paratick and test it in more diverse scenarios, focusing on high-performance I/O applications. Eventually, we aim to propose a patch based on paratick for the mainline Linux kernel.

ACKNOWLEDGMENTS

We thank the reviewers for their insightful feedback. This work is partially funded by the US National Science Foundation through grant CCF 1617749.

REFERENCES

[1] 2016. *[V4,4/4] Utilize the vmx preemption timer for tsc deadline timer.* https://patchwork.kernel.org/project/kvm/patch/1465852801-6684-5-git-send-email-yunhong.jiang@linux.intel.com/

[2] 2019. Timer Interrupt Sources. https://wiki.osdev.org/Timer_Interrupt_Sources

[3] 2021. *Open-Source, Automated Benchmarking.* https://www.phoronix-test-suite.com/

[4] 2021. *torvalds/linux.* https://github.com/torvalds/linux

[5] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices* 41, 11 (2006), 2–13.

[6] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh. 2014. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, 394–405.

[7] Mohit Aron and Peter Druschel. 2000. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 197–228.

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[9] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems.* 26–35.

[10] Christian Bienia and Kai Li. [n.d.]. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *MoBS 2009.*

[11] Timothy Broomhead, Laurence Cremean, Julien Ridoux, and Darryl Veitch. 2010. Virtualize Everything but Time.. In *OSDI*, Vol. 10. 1–6.

[12] Mauro C. Chehab and Julia Lawall. 2020. NO HZ: Reducing scheduling-clock ticks. *Linux Kernel Source Tree* (July 2020). https://github.com/torvalds/linux/blob/master/Documentation/timers/no_hz.rst

[13] Luwei Cheng, Jia Rao, and Francis CM Lau. 2016. vscale: Automatic and efficient processor scaling for smp virtual machines. In *Proceedings of the Eleventh European Conference on Computer Systems.* 1–14.

[14] Jonathan Corbet. 2015. Reinventing the timer wheel. https://lwn.net/Articles/646950

[15] Xiaoning Ding, Phillip B Gibbons, Michael A Kozuch, and Jianchen Shan. 2014. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14).* 73–84.

[16] Xiaoning Ding and Jianchen Shan. 2015. Diagnosing Virtualization Overhead for Multi-threaded Computation on Multicore Platforms. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom).* IEEE, 226–233.

[17] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480.

[18] Sandeep D'Souza and Ragunathan Rajkumar. 2018. QuartzV: Bringing Quality of Time to Virtual Machines. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 49–61.

[19] Yoav Etsion, Dan Tsafrir, and Dror G Feitelson. 2003. Effects of clock resolution on the scheduling of interactive and soft real-time processes. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems.* 172–183.

[20] Yoav Etsion, Dan Tsafrir, and Dror G Feitelson. 2003. Effects of clock resolution on the scheduling of interactive and soft real-time processes. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems.* 172–183.

[21] Thomas Friebel and Sebastian Biemueller. 2008. How to deal with lock holder preemption. *Xen Summit North America* 164 (2008).

[22] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. 2002. Supporting time-sensitive applications on a commodity OS. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 165–180.

[23] Ahmad Golchin. 2017. *Control based tickless scheduling.* Ph.D. Dissertation.

[24] Intel. 2019. Intel(R) RDT Software Package. https://github.com/intel/intel-cmt-cat.

[25] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2018. Scaling Guest OS Critical Sections with eCS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).* 159–172.

[26] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14).* 61–72.

[27] Jiannan Ouyang and John R Lange. 2013. Preemptable ticket spinlocks: Improving consolidated performance in the cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments.* 191–200.

[28] Jiannan Ouyang, John R Lange, and Haoqiang Zheng. 2016. Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs. *ACM SIGPLAN Notices* 51, 7 (2016), 17–23.

[29] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 2008. 30 seconds is not enough! A study of operating system timer usage. *ACM SIGOPS Operating Systems Review* 42, 4 (2008), 205–218.

[30] Rusty Russell. 2005. Unreliable Guide To Hacking The Linux Kernel. https://www.kernel.org/doc/htmldocs/kernel-hacking/index.html

[31] Stijn Schildermans, Kris Aerts, Jianchen Shan, and Xiaoning Ding. 2020. Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency. *ISPA-BDCloud-SocialCom-SustainCom 2020* (2020), 76–83.

[32] Stijn Schildermans, Jianchen Shan, Kris Aerts, Jason Jackrel, and Xiaoning Ding. 2021. Virtualization Overhead of Multithreading in X86 State of the Art & Remaining Challenges. *IEEE Transactions on Parallel & Distributed Systems* 01 (2021), 1–1.

[33] Jianchen Shan, Xiaoning Ding, and Narain Gehani. 2016. APPLES: Efficiently handling spin-lock synchronization on virtualized platforms. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2016), 1811–1824.

[34] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. 2007. Getting maximum mileage out of tickless. In *Proceedings of the Linux Symposium*, Vol. 2. Citeseer, 201–207.

[35] Houjun Tang, Quincey Koziol, Suren Byna, John Mainzer, and Tonglin Li. 2019. Enabling Transparent Asynchronous I/O using Background Threads. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW).* IEEE, 11–19.

[36] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. 2015. A comprehensive implementation and evaluation of direct interrupt delivery. *Acm Sigplan Notices* 50, 7 (2015), 1–15.

[37] VMware. 2011. Timekeeping in VMware Virtual Machines.

[38] C-Q Yang and Barton P Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *8th International Conference on Distributed.* IEEE Computer Society, 366–367.

[39] Matıas Zabaljáuregui. 2008. Hardware assisted virtualization intel virtualization technology. *accessed at linux. linti. unlp. edu. ar/images/f/f1/Vtx. pdf* (2008), 1–54.