Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges

Virtualization Overhead of Multithreading in X86

State of the Art & Remaining Challenges

Stijn Schildermans, Jianchen Shan, Kris Aerts, Jason Jackrel and Xiaoning Ding

Abstract—Despite great advancements in hardware-assisted virtualization of the x86 architecture, certain workloads still suffer significant overhead. This work dissects said overhead in the context of multi-threading. We describe the state of the art, pinpoint challenges and suggest improvements, aiming to provide a valuable reference to developers and users of virtualization systems alike. We study the virtualization overhead of the PARSEC and SPLASH2X multithreaded benchmarks in a variety of scenarios using a state-of-the-art system. Through controlled experiments, source code analysis and literature review, we quantify the virtualization overhead multithreading still induces and link it to its root causes, after which we suggest possible mitigation strategies. Multithreading still induces high virtualization overhead, mainly caused by synchronization, spinning at user level and NUMA management. The overhead is diverse in nature and embodiment as it is a function of many system and workload properties. System-level solutions are feasible, but often imply difficult trade-offs. Systematic workload optimization is a promising alternative.

Index Terms—Multi-threading, virtualization, overhead, performance, guidelines, classification.

1 Introduction

Virtualization is known to have many benefits, as exemplified by the blooming popularity of cloud computing [1]. However, these benefits come at a cost, mainly in the form of efficiency and performance losses. Identifying and mitigating said losses is a long-standing challenge for researchers at all system layers [2], [3], [4], [5], [6]. Thanks to these efforts, most virtualized workloads can presently achieve near-native performance [7]. Some types of workloads however deviate from this trend. Prominent examples of such workloads are multi-threaded applications [8]. Because deploying these applications in a virtualized setting is becoming the norm in this era of cloud computing, HPC, IoT and big data, we deem optimizing their performance in this context paramount. With this work we aim to contribute thereto by providing an overview of the state of the art regarding hardware-assisted virtualization of multithreading in x86, identifying major outstanding issues and exploring to what extent these can be addressed.

A major motivation for this work is our observation that research regarding virtualization of computation-intensive workloads is losing momentum. We speculate that considerable progress in this field over the past decade and the emergeance of containerization are main drivers of this trend. We find both of these reasons unfounded. Virtualization is still widely used in industry due to its distinct benefits over containerization such as far fewer security risks and increased flexibility [9]. Furthermore, this paper will show that virtualization overhead is still far from eliminated in the studied context, and that accepted solutions to some issues have much room for improvement. We thus aim to reinvigorate research into hardware virtualization for x86, in particular for multi-threaded applications.

We specifically target multi-threaded applications for 4 reasons. Firstly, literature lacks a systematic study regarding the issues arising from virtualizing this application type, in contrast to many others [10], [11], [12]. While studies related to our goal are plentiful [13], [14], [15], their scope is limited to specific phenomena. Secondly, thread-coordination with minimal VMM intervention is conceptually challenging and demands much research attention [8]. Thirdly, multi-threaded applications are by nature very sensitive to overhead. As we will show in §5.2, a small amount of overhead may cause significant performance degradation. Lastly, virtualization technology has advanced considerably in recent years, addressing many classic problems affecting multi-threaded workloads, such as lock-holder preemption [16]. We aim to assess to what extent these advancements are successful and which challenges remain.

1.1 Methods

The paper first formally defines virtualization overhead. Based on this definition, we present a quantitative analysis of virtualization overhead for multi-threaded workloads based on controlled experiments using state-of-the-art hardware-assisted x86 virtualization techniques. We use common performance profiling tools to collect our results and verify them through source code analysis and literature review. Besides plainly describing the overhead, we provide a deeper understanding thereof by linking it to its conceptual causes. Lastly, we reflect on promising directions for future work that may mitigate said causes.

While we can not guarantee that all our findings are universally applicable, we cover a wide variety of system configurations and workloads to minimize the threats to validity inherent to empirical work such as this. Moreover, we reflect on how our findings would translate to scenarios we did not cover explicitly (e.g. other hardware/hypervisors).

- *S. Schildermans and K. Aerts are with the Department of Computer Science, KU Leuven, Diepenbeek, Limburg, 3590, Belgium. E-mail: {kris.aerts, stijn.schildermans}@kuleuven.be*
- *J. Shan and J. Jackrel are with the Department of Computer Science, Hofstra university, Heampstead, NY, 11549, USA. E-mail: jianchen.shan@hofstra.edu / jjackrel1@pride.hofstra.edu*
- *X. Ding is with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, 07102, USA. E-mail: xiaoning.ding@njit.edu*

1.2 Main Findings & Contributions

• We propose a definition for virtualization overhead that explicitly divides it into internal system effects and external application effects. This allows overhead to be described accurately and unambiguously.
• With the latest virtualization support, overhead imposed on individual threads is low. For sequential applications, overhead is mainly incurred by handling I/O.
• Virtualization overhead for multi-threaded applications has been significantly reduced in recent years thanks to various advancements in virtualization technology.
• Multi-threaded computations still suffer significant virtualization overhead, especially when the system is overcommitted. Thus, further improvements are desirable.
• For multi-threaded applications, there can be a large divergence between internal virtualization overhead and externally observed performance degradation when the overhead is incurred on the critical path.
• Most virtualization overhead incurred by multi-threaded applications is caused by interaction between threads, in the form of data sharing (especially in NUMA systems) and synchronization (especially spinning at user level and blocking synchronization).
• Most multi-threaded workloads benefit from being consolidated using virtualization. Some even consume less resources when consolidated.
• Abstraction of underlying NUMA architectures still poses an issue for multi-threaded applications. Modern techniques to optimize vCPU placement are still lacking.
• Remaining overhead is hard to tackle at the system level. Application-level solutions are, however, promising and understudied as of now.

2 Background

2.1 Hardware-assisted Virtualization

Virtual machines (VMs) allow execution environments to remain identical when underlying implementations, systems, or hardware change. VMs range from single-process (e.g. JVM) to whole system VMs running a complete OS [17].

Most guest instructions are executed directly on the hardware [18]. Privileged and sensitive operations must however be handled differently because they can break the virtualization barrier. Today, hardware-assisted virtualization has become the dominant technique to implement this in x86. This involves the CPU trapping to a dedicated piece of software, a hypervisor (VMM) (e.g., Xen [19], vSphere [20], and KVM [21]), when it detects a sensitive operation. The VMM handles the traps and coordinates VMs. Intel and AMD both implement this technology in most of their CPUs (resp. VT-x and AMD-V [16], [22]).

Contrary to other methods, hardware-assisted virtualization allows the guest to run in the CPU's privilege ring 0, allowing it to execute most privileged instructions without costly traps to the hypervisor, which itself runs in a dedicated *VMX root mode*, with full control over the system. The CPU traps to root mode when the VM carries out a sensitive operation, allowing the VMM to intervene. Such traps are called *VM exits*. System administrators can control to a large extent which operations trigger VM exits [23].

2.2 Virtualization Performance Issues

It is well-known that applications tend to show lower performance when virtualized for various reasons. Below we elaborate on the most important of these, as well as the techniques already adopted to mitigate the issues:
• Multiple VMs often share hardware resources. Due to inefficient resource management policies in the VMM or unmanaged contention between VMs, applications may be unnecessarily starved of resources such as CPU, cache or memory. Many efforts have been made to improve this (e.g. memory deduplication [24], Intel RDT [25]).
• At the VMM level, emulation of sensitive operations is still a major cause of performance degradation for certain workloads. While some virtualization techniques (i.e. paravirtualization) avoid this cost, doing so has other drawbacks such as reduced flexibility [26].
• I/O operations, such as accessing I/O ports, DMA, and interrupts, are all privileged and trapped. Additionally, for high bandwidth I/O devices, extra data needs to be copied to the VMM. Techniques mitigating this include paravirtualization (e.g. paravirtualized drivers sharing I/O buffers between VM and VMM) [27] and hardware assistance [11].
• In virtualized systems, guest memory accesses have to be translated to VMM-managed machine addresses. Two techniques are common for this, namely VMM-level *shadow page tables* [28] and hardware-level *nested paging* [29]. Both techniques are still in use, and cause (limited) overhead in their own ways [26]. Neither of them is universally superior.
• Spinning synchronization is often used to coordinate short critical sections in OS kernels. When the hardware is overcommitted however, the VMM may deschedule a vCPU holding a spin lock, causing the vCPUs waiting for that lock to waste cycles. This is known as *lock holder preemption (LHP)* [13]. Various related problems have been identified (e.g. *lock-waiter preemption* [30]). Several approaches have been proposed to mitigate such issues. Hardware extensions that trigger a VM exit when a vCPU executes excessive amounts of `PAUSE` instructions -indicating spinning- are already widely adopted (*Pause Loop Exiting (PLE)* for Intel [16] and *Pause Filter (PF)* for AMD [22]).
• VCPUs holding a blocking-based lock can also be descheduled by the VMM while other threads are waiting for it. When a thread blocks on such a lock and the guest has no more useful work to do, it will issue a `HLT` instruction, triggering a VM exit and running the VMMs scheduler excessively. This is known as the *blocked-waiter wakeup (BWW)* problem [15]. Some VMMs implement *halt polling* to help mitigate this. When a VM exit due to `HLT` occurs, the VMM will spin for some time before executing its scheduler or halting the pCPU, hoping the vCPU is woken up by the guest. If so, vCPU execution is immediately resumed.
• Usually the guest is unaware of the exact physical hardware configuration. This can decrease e.g. cache and memory performance. Particularly for NUMA systems this is an issue, since NUMA-unaware scheduling can greatly increase memory and synchronization latency [1]. Several solutions to this problem have been developed, such as NUMA-aware VMM schedulers [31], dedicated VMM-level NUMA locality managers [32] and exposing the NUMA-architecture to the guest [33].

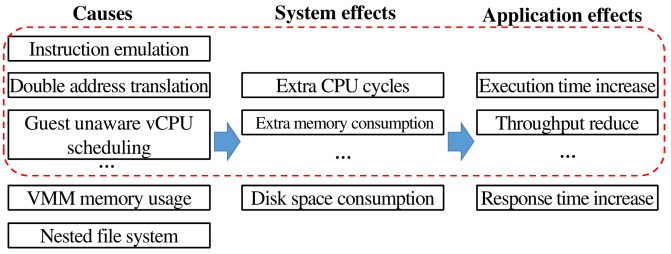


Fig. 1. Breakdown of potential virtualization effects and their causes. The dotted box circles the scope of the paper.

3 Defining Virtualization Overhead

Thoroughly analyzing virtualization overhead requires unambiguously defining it. Most literature only measures performance degradation when executing an application in a VM (e.g., the increase in wall-clock execution time). However, we argue that the toll on the entire system should be considered. Fig. 1 provides a breakdown of this toll.

Based on fig. 1, virtualization overhead consists of a set of *system effects*, viewed internally from the host's perspective, which translate to a set of *application effects*, viewed externally from the user's perspective. These are not necessarily correlated. For example, when a server is not overloaded, I/O operations may be offloaded to redundant cores, not slowing down the workloads in the VMs. We argue that this 'concealed' cost is important for 2 reasons: firstly, public cloud environments are evolving towards charging consumers for CPU cycles used by their VMs [34], rather than VM uptime (e.g. serverless environments [35]). Secondly, 'concealed cost' may become 'visible' after all in certain scenarios (e.g. under heavy server load). Below we provide a formal definition of both effect classes from fig. 1.

3.1 System Effects

Any excess resource usage caused by virtualization (cycles, memory, bandwidth,...) is a system effect. However, we are only interested in system effects due to multithreading. As this is a purely computational concept, the main resource of interest to us is the CPU. While other metrics such as memory usage may be important, from a pragmatic perspective this is only an issue when they bottleneck the system. This will however be reflected by increased CPU cycles. Thus, we define the system effects of virtualizing multithreading in terms of CPU cycles as *reduced resource efficiency*, $\delta\eta_r$:

Let $C_p(W, P(S_w))$ be the CPU cycles used by workload W on physical system $P(S_w)$, with S_w all settings for P. Let $C_v(W, V(S_w), P(S_v))$ be the system cycles used by W on a virtual machine $V(S_w)$ with the same settings, hosted on a system $P(S_v)$. Then $\delta\eta_r = \frac{C_v(W,V(S_w),P(S_v)) - C_p(W,P(S_w))}{C_p(W,P(S_w))}$.

S_v includes all system settings only visible to the VMM, e.g. the VMM used, concurrent VMs, etc. S_w reflects all settings observable within the (guest) OS, e.g. concurrent applications, CPU count, etc. Note that it is almost impossible to guarantee S_v and S_w remain constant between executions due to non-deterministic aspects of the system (e.g. interrupts, background processes,...). To reduce the variance in S_v and S_w to negligible levels, experimental results should always be averaged over many iterations.

3.2 Application Effects

Like system effects, application effects encompass different metrics such as latency, throughput, etc. The main metric of interest however is wall-clock execution time. Other metrics indirectly translate thereto (e.g. reduced system throughput increases execution time). Analogously to reduced resource efficiency, we can thus define *reduced temporal efficiency*, $\delta\eta_t$, as the increased time needed to execute a workload in a VM. One addition must be made though. Since wall-clock time is measured externally and S_v may include temporally multiplexing resources between $V(S_w)$ and other tasks, we must take the effective resources available to the VM into account. We thus essentially separate the effects of resource sharing from those of virtualization. Based on §3.1, we use the amount of available CPU time as a proxy for resources in general. This yields the following definition for $\delta\eta_t$:

$$\delta\eta_t = \frac{t_v(W,V(S_w),P(S_v)) \times \gamma_v - t_p(W,P(S_w)) \times \gamma_p}{t_p(W,P(S_w)) \times \gamma_p}$$

with t_p and t_v the real times for executing the workload in resp. the physical and virtual environments and γ_p and γ_v the ideal effective CPU count available to the workload given S_w and S_v in each resp. environment.

4 Experimental Setup

Below we describe the main experiments for this study. We carefully designed these so that our findings are as general as possible. Wherever this generality is not guaranteed -as is often the case with empirical work- we reason on how our results would translate to other prevalent contexts in §6.

4.1 System

The CPU is by far the most important hardware component in hardware-assisted virtualization. Since Intel dominates the corporate x86 server CPU market, with AMD having a market share of only 8%, we focus on Intel VT-x for our experiments [36]. However, results can be safely generalized to AMD-v, since it is nearly identical to VT-x [16], [22]. To our knowledge, no studies suggest a notable performance difference in any regard between these technologies.

Concerning hypervisors, four players dominate with a combined market share of over 95%: VMWare, Hyper-v, Xen, and KVM [37]. Unfortunately, the most popular of these -VMWare and Hyper-v- are closed source. This means we can not verify empirical results by analyzing VMM source code for these VMMs. We therefore limit our detailed analysis to Xen (HVM) and KVM. Because previous studies have shown that KVM is in general by a narrow margin slower than Xen for CPU-bound workloads [38], we pick KVM for our experiments in an effort to err on the side of caution, minimizing the risk of our results being overly optimistic in a generalized context.

For the guest OS we opt for Linux since it is by far the most popular server OS, with the only noteworthy competitor being Windows [39]. The latter is however closed-source, making analysis of results again difficult. Moreover, we intuitively expect the guest OS not to be a major contributor to virtualization cost, which justifies only using 1 guest OS.

We create environments with 4, 8, 16, 32 and 64 CPUs. To emulate $P(S_w)$ from §3, we use `taskset`[1] to limit the

1. https://linux.die.net/man/1/taskset

usable CPUs. To emulate $V(S_w)$, we create VMs with said vCPU counts and 64 GB of memory. The vCPUs were laid out in a way preventing the vCPU stacking problem [40].

Concretely, the host system is a HPE ProLiant DL385 Gen10 server with 4 Intel Xeon Gold 6138 20-core processors and 256GB of memory. Hyperthreading was disabled, as were C states deeper than C1 to prevent performance degradation due to CPU power management [41]. Ubuntu Server 18.04.2 (kernel 4.15) is the OS for both the host and the guest, as it is the most recent version of one of the most popular Linux distributions at the time of writing [42]. Because each CPU only has 20 cores, we spread the CPUs for the 32- and 64-core environments equally over resp. 2 and 4 sockets. This allows us to study how virtualization cost is influenced by NUMA architectures as an added benefit.

4.2 Workloads & Measurement

Since this study focuses on multithreading as a concept, multithreaded, computation intensive benchmarks employing minimal I/O are a natural workload choice. We prefer using a well-rounded benchmark suite over hand-picking or devising arbitrary programs. We found that the PARSEC 3.0 and SPLASH2X benchmark suites fit our requirements perfectly [43]. These suites contain 26 multithreaded, computation-intensive workloads designed to cover a wide range of real-world tasks ranging form games to scientific computation, maximizing generalizability of results.

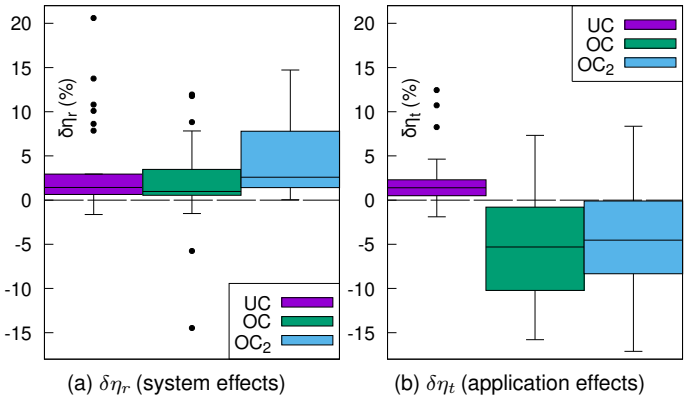
All benchmarks were compiled using `pthreads` and run with their 'native' inputs [43]. The level of parallelism is set equal to the number of CPUs for each test. We always pre-warmed the OS buffer cache to minimize I/O operations. We take the average of 10 iterations as our result. We use the `perf`[2] profiling tool for all measurements.

4.3 Scenarios

Because certain forms of overhead only appear when multiple VMs share resources [13], we conduct all experiments in 2 scenarios: *undercommitted (UC)* and *overcommitted (OC)*. We launch resp. 1 and 2 identical VMs running a benchmark instance on the same pCPU set. By using identical workloads we avoid unfair resource allocation, which is known to be an issue for synchronization-heavy workloads [44]. When both VMs demand all available resources, each will receive 50% thereof. Thus, $\gamma_v = \frac{\gamma_p}{2}$. VMM cycles can also be split equally between VMs, so that $C_v = \frac{C_{VMM}}{2} + C_{VM} = \frac{C_{sys}}{2}$. Note that when the VMs would not be running identical workloads, these measurements would be much more complicated.

Since in §3 $P(S_w) \cong V(S_w)$, C_p and t_p refer to undercommitted native execution, even when S_v includes overcommitting the system. This is conceptually sound, since multiplexing system resources between $V(S_w)$ and other tasks is opaque to the VM and thus a virtualization effect from the perspective of the workload. On the other hand, this intertwines the effects of virtualization and hardware consolidation. To address this, we supplement the UC and OC data sets discussed above with the data set 'overcommitted base 2' (OC_2), which directly compares C_v and t_v for 2 concurrent VMs each running one instance of W to C_p and t_p when executing 2 concurrent instances of W on $P(S_w)$.

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



(a) $\delta\eta_r$ (system effects) (b) $\delta\eta_t$ (application effects)

Fig. 2. Box plots of virtualization overhead for the sequential versions of all PARSEC and SPLASH2X workloads, aggregated for each scenario.

5 Results

5.1 Sequential Applications

We first briefly analyze virtualization overhead for sequential applications to compare it to their multithreaded counterparts. Fig. 2 shows the aggregate results for sequential executions of all PARSEC and SPLASH2X workloads.

As fig. 2 shows, virtualization overhead has been minimized for sequential workloads. On average, both $\delta\eta_r$ and $\delta\eta_t$ are negligible. Some outliers can be observed however. Detailed analysis reveals that these are attributable to I/O. This is a well-known issue, as described in §2.2.

Generally, $\delta\eta_r > \delta\eta_t$ in fig. 2. In the OC scenario, $\delta\eta_t$ is even negative. We found that QEMU is responsible for this, as it has to handle write-backs of newly generated data (reads come from the pre-warmed OS buffer cache). This consumes up to 20% of the CPU cycles. Because QEMU runs on a separate thread in parallel with the VMs, this does not increase $\delta\eta_t$. On the contrary, this effect results in a negative $\delta\eta_t$ in the OC scenario since a vCPU from the second VM may run while the first is waiting for QEMU.

5.2 Multithreaded Applications

We collected virtualization overhead for multithreaded applications analogously to §5.1. Fig. 3 shows the results with a separate set of bars for each vCPU count.

Fig. 3 shows that also for multithreaded applications, $\delta\eta_t$ is limited in general. In the OC scenario, it is even strongly negative; increasingly so as the vCPU count increases. Firstly, this is caused by processing I/O in the background, as described in §5.1. Secondly, the 2 instances of the benchmark that share each pCPU can compensate for each other's idle time. When a vCPU starts to idle in the UC scenario, the pCPU is also idle. In the OC scenario however, a vCPU from the other VM can be scheduled, thus increasing system throughput. This is confirmed by the OC_2 data set, for which $\delta\eta_t$ is positive as in a native setting this consolidating effect also occurs.

Fig. 3 also shows that multithreaded applications still suffer high virtualization overhead compared to sequential ones. Overhead tends to greatly increase with vCPU count, indicating that mitigating it will only gain importance as time goes on, since VMs tend to become ever larger in size.

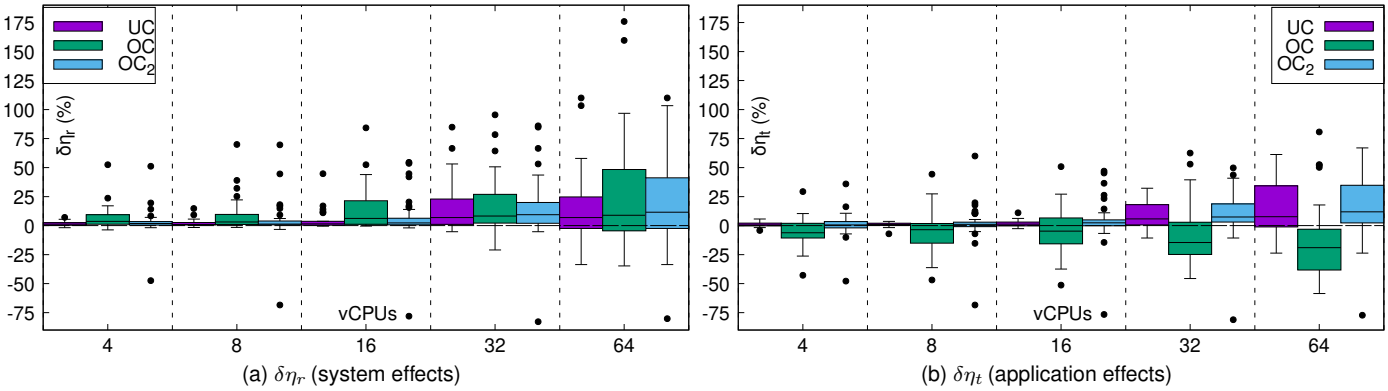


Fig. 3. Box plots of metrics of interest for multi-threaded executions of all PARSEC and SPLASH2X workloads. Results are shown separately for each vCPU count. Overcommitted results are measured in 2 ways, as described in §4. Results for all benchmarks are aggregated for each scenario.

However, we are pleased to find that great advancements have been made in the past few years. For example, a study from only 5 years ago found that the performance of *Dedup* could be degraded by more than 500% when the system is overcommitted [8], much more than any value in fig. 3.

When comparing figures 2 and 3, the variance between benchmarks appears to be much greater for multithreaded executions than for sequential ones. For some benchmarks $\delta\eta_r$ is strangely negative, while for others it may be over 150%. To gain a deeper understanding of these results, we provide a detailed breakdown of the multithreaded executions in fig. 4, showing the average and maximum $\delta\eta_t$ (4a) and $\delta\eta_r$ (4b) for each benchmark for all vCPU counts combined with overlapping bars. Fig. 4b breaks each of the overlapping bars down into 2 parts stacked on top of each other: cycles spent in the guest and host, resp.

Fig. 4 provides several insights. Firstly, the OC_2 data set explains why $\delta\eta_r < 0$ for some benchmarks in the OC scenario (e.g. *FFT*, *Radiosity*, *s.Raytrace*). Namely, overcommitting has a positive effect on η_r in a native setting as well. This is thus an effect of consolidation rather than virtualization. The main causes are the following:

• **Reduced lock contention**: As the system is overcommitted, the effective CPU utilization of individual benchmarks is lower. As less threads are competing for the same spin locks, less cycles are wasted spinning;
• **NUMA management**: When the system is overcommitted, the scheduler can do a better job of balancing the workload between different NUMA nodes, thus reducing memory latency. This is discussed in detail in §6.1.
• **Reduced idling**: When a CPU runs out of work, the OS performs several operations to prepare it to idle. We explain this in detail in §6.2.1. When the system has more work, it is less likely to start idling, thus eliminating these operations.

Lastly, $\delta\eta_r$ and $\delta\eta_t$ may be wildly divergent for multithreaded workloads in contrast to their sequential counterparts according to figures 2 and 3. To better understand this, we define the *overhead impact factor* $\omega = \frac{1+\delta\eta_t}{1+\delta\eta_r}$ as a measure of the correlation between system and application effects. For multithreaded applications, the variance in ω ($\sigma\omega$) is very high. For example, for *Bodytrack*, UC $\omega \approx 1.1$, while for *Ocean CP*, OC $\omega \approx 0.6$. This suggests that overhead may vary in nature depending on the workload.

The main reason for the high $\sigma\omega$ in fig. 4 is that the runtime of a multithreaded application is determined by its critical path [45]. When $\delta\eta_r$ is located mostly on the critical path, $\delta\eta_t$ increases drastically. Otherwise, $\delta\eta_r$ may have little to no effect on $\delta\eta_t$. To illustrate this, we collected the cycles spent by each (v)CPU for *Bodytrack* and *Ocean CP*, 64 vCPUs, UC. Fig. 5 shows the distribution of the native and virtualized cycles by vCPU ID, normalized to native.

Fig. 5 shows that system-level overhead is distributed very differently between vCPUs for *Bodytrack* and *Ocean CP*. None of the vCPUs show much overhead for *Bodytrack*, except for 1. It is likely other vCPUs will at some point have to wait for this overhead-heavy vCPU since it has so much work, thus slowing down the entire application. For *Ocean CP*, the distribution of extra work is much more egalitarian. Because of this, many of the extra cycles are likely not part of the critical path, yielding a much smaller ω.

6 Virtualization Overhead Breakdown

The large variance in $\delta\eta_r$, $\delta\eta_t$ and ω between benchmarks in §5.2 suggests that multiple causes are responsible, warranting detailed analysis. Because fig. 4 reveals patterns in overhead profiles between benchmarks, we begin by categorizing them based on said profiles. Since $\delta\eta_r$ and $\delta\eta_t$ are not strongly correlated and $\delta\eta_t$ represents merely the external symptoms of $\delta\eta_r$, we focus only on the latter as a guiding metric for this categorization. Some benchmarks exhibit characteristics of several overhead profiles and were therefore added to multiple categories.

• **Negligible overhead (OH)**: *Barnes*, *Ferret*, *FFT*, *FMM*, *Freqmine*, *LU NCB*, *parsec.Raytrace*, *Radiosity*, *splash2x.Raytrace*, *Swaptions*, *Water NSquared* and *Water Spatial*.
• **High guest OH**: *Blackscholes*, *Canneal*, *Fluidanimate*, *Ocean CP*, *Ocean NCP* and *Radix*.
• **High host OH**: *Bodytrack*, *Dedup*, *Facesim*, *Vips* and *Volrend*.
• **High OC OH**: *LU CB*, *Streamcluster*, *Vips*, *Volrend*, *X264*.

Below we discuss each of the categories defined above in detail. Because figures 3 and 4 indicate that overhead varies severely between VM sizes, we start each analysis by breaking the overhead down for each VM size in the most relevant scenario, after which we reason about the causes and reinforce our conclusions with empirical evidence. We carefully consider the generality of our findings.

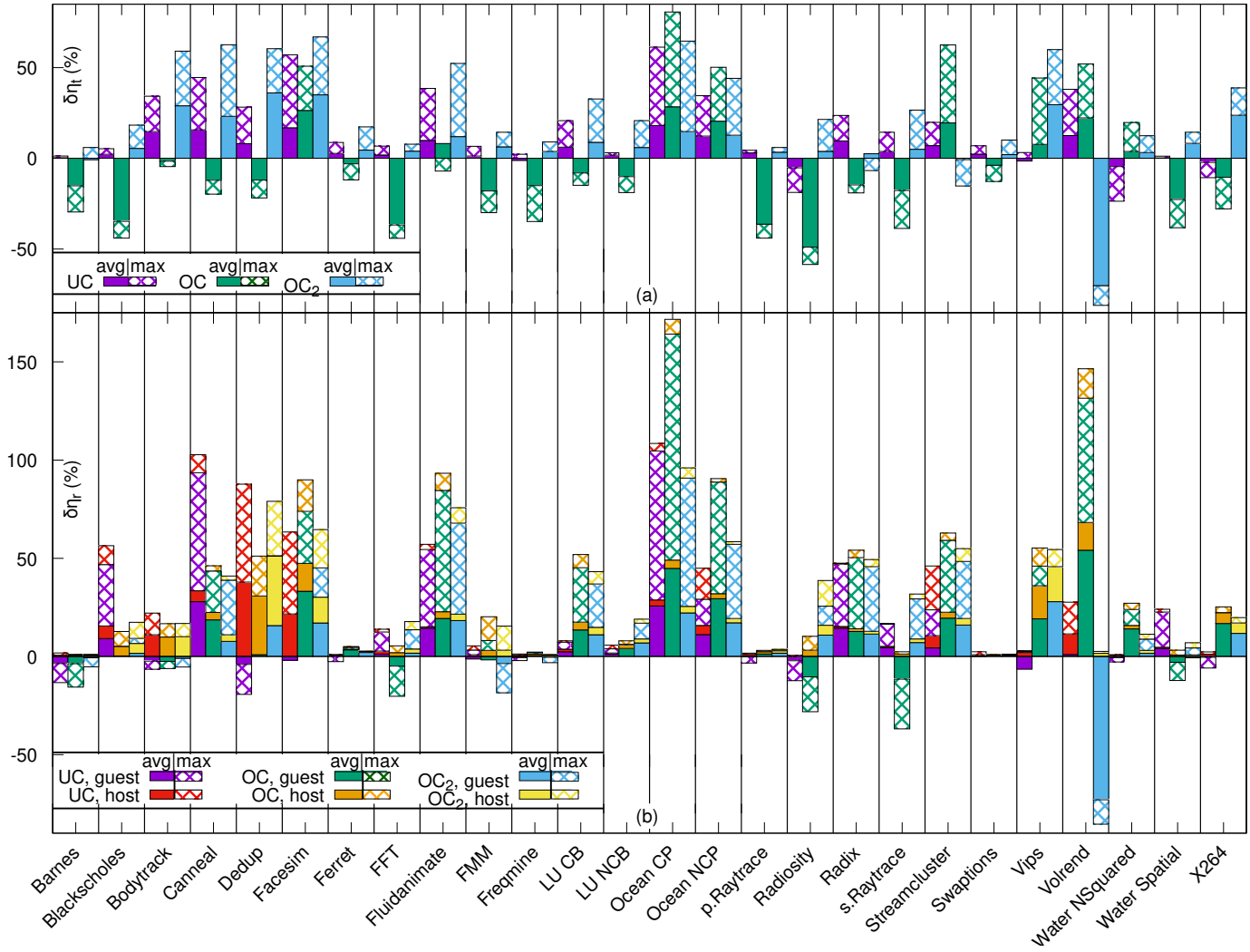


Fig. 4. Average and maximum $\delta\eta_t$ (a) and $\delta\eta_r$ (b) for the studied vCPU counts, displayed separately for each benchmark with overlapping bars.

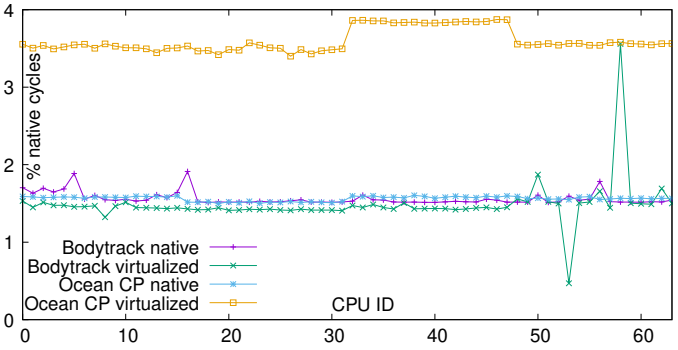


Fig. 5. Distribution of cycles over (v)CPUs for the 64 (v)CPU variants of *Bodytrack* and *Ocean CP*, normalized to native so that $\sum_{C=0}^{63} P(C) = (\delta\eta_r + 1)(\times 100\%)$, with C a particular CPU ID.

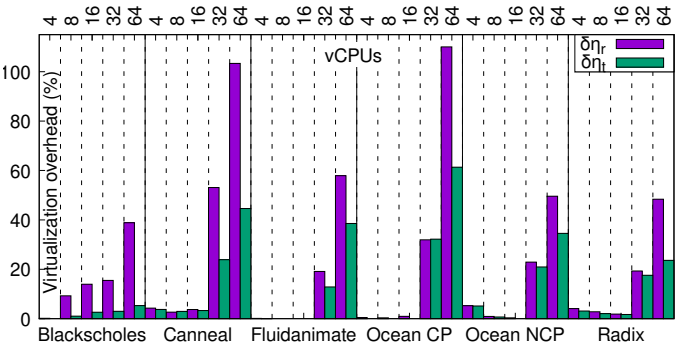


Fig. 6. Breakdown of virtualization overhead for the benchmarks with high guest overhead per studied vCPU count in the UC scenario.

6.1 High Guest Overhead

Several benchmarks with high guest overhead display most overhead in the UC scenario in fig. 4. While some show higher OC overhead, the OC_2 data is similar to UC, indicating that even on physical systems, overcommitting adds overhead. The increase in OC overhead is thus due to resource consolidation rather than virtualization. We therefore conclude that analyzing the UC scenario is sufficient for this category of benchmarks. Fig. 6 shows the results.

In fig. 6, overhead is negligible for vCPU counts below 32, after which it increases enormously. Since from 32 vCPUs we use multiple sockets, NUMA may be the culprit. Namely, memory-intensive applications may often access data on remote NUMA nodes. In a VM, the guest scheduler lacks NUMA information, preventing it from optimizing locality like it would natively. For computation-intensive workloads like ours, cycles per instruction (CPI) can prove this hypothesis by indicating memory latency [46], as shown in fig. 7.

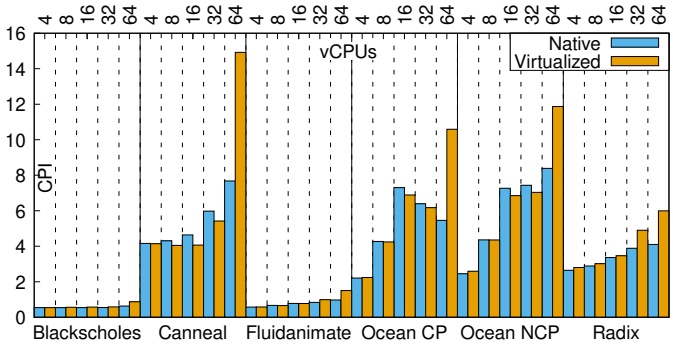


Fig. 7. CPI for the benchmarks of interest per vCPU count, UC.

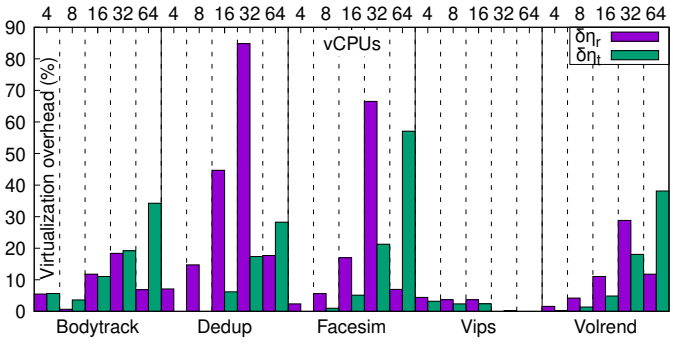


Fig. 8. Breakdown of virtualization overhead for the benchmarks with high host overhead per vCPU count in the UC scenario.

Fig. 7 verifies our intuition. Overhead is highest for the benchmarks with the highest CPI, being the most memory-intensive benchmarks. For native executions, CPI increases slightly with CPU count. When virtualized, this increase is much more pronounced, particularly for 64 vCPUs. *Ocean CP* is the only exception. Detailed analysis shows that this benchmark is bottlenecked by memory bandwidth. When more CPU sockets are used, available bandwidth increases, improving performance despite increased memory latency.

For all benchmarks in fig. 6, ω is low. The reason for this is that performance-critical data tends to be accessed often and thus cached. Only data that is rarely used is fetched from main memory, which is usually input for worker threads and therefore not directly on the critical path.

Abstraction of the underlying system is a core concept of virtualization, implying that the above issue is independent of the virtualization technology used. Rather, it depends on the host system $P(S_v)$. All popular virtualization platforms are known to struggle with NUMA locality [4], [47].

6.2 High Host Overhead

Most benchmarks in this category suffer most in the UC scenario. Those that do not (*Vips* and *Volrend*) are also included in the 'high overcommitted overhead' category. To avoid duplicate results, we only break down the overhead in detail for the UC scenario here. Fig. 8 shows the results.

Fig. 8 is interesting. $\delta\eta_r$ rises until 32 vCPUs, but drops severely at 64. $\delta\eta_t$ however keeps rising for all benchmarks except *Vips*. ω thus varies greatly between benchmarks and vCPU counts. It is obvious that a deeper analysis is necessary. Since any host operations are preceded by a VM exit in hardware-assisted virtualization, we break down the host-level CPU cycles by VM exit reason in fig. 9. We show both the UC and OC scenarios to include meaningful results for *Vips* and *Volrend*.

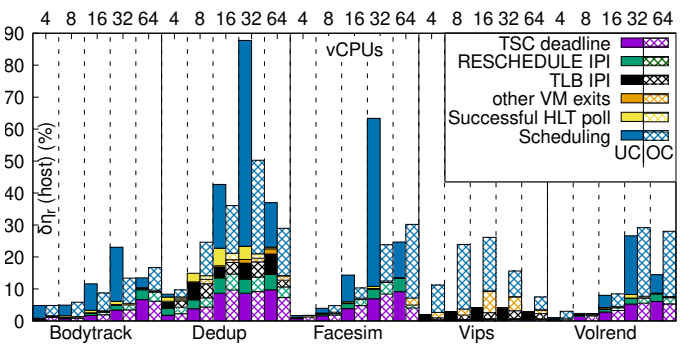


Fig. 9. Breakdown of host cycles for the benchmarks with high host overhead into their main causes per vCPU count.

Fig. 9 explains the variance in ω observed in fig. 8. The strange pattern for $\delta\eta_r$ is exclusively attributable to scheduling. When scheduling cycles are ignored in fig. 8, one observes a consistent, high ω. This makes sense, since in the UC scenario, VMM-level scheduling only occurs when the VM voluntarily yields the vCPU. Therefore, host-level scheduling is rarely on the critical path. Most other VM exits are caused by the VM performing sensitive operations. Many of these are by nature highly likely to be on the critical path, thus yielding a high ω. Below, we discuss fig. 9 in detail in terms of the high-level overhead causes.

6.2.1 Blocking Synchronization

Blocking synchronization is prevalent in multithreaded applications. VMM intervention is usually not necessary in this process, as it is mostly implemented in user space. There are however 3 exceptions to this rule:

- When a thread blocks and no other work is available for the (v)CPU, the OS executes the `HLT` instruction. This generates a VM exit. The VMM then usually schedules another vCPU, if available. Because scheduling is expensive, KVM implements an optimization called *halt polling* [48]: The host first polls for a dynamically determined amount of time before scheduling. If the vCPU is woken up by the guest kernel during this time, it is immediately rescheduled.
- When a contended lock is released, usually one waiting thread is woken up. If there are any idle CPUs, the kernel sends one of them a *RESCHEDULE* inter-processor interrupt (IPI). On the receiving (v)CPU, the scheduler is invoked and the newly awoken thread is run [44]. Sending an IPI requires writing to the `ICR` MSR, which triggers a VM exit.
- Linux updates the global system time through the *scheduler tick*: periodic per-CPU timer interrupts driven by the CPU's *time stamp counter (TSC)* (250 Hz for Ubuntu 18.04). When a thread blocks and no runnable tasks are available for the (v)CPU, an *idle governor* runs to predict heuristically how long the CPU will likely be idle. If the predicted idle time is sufficiently long, the idle governor reduces the tick frequency to 1 Hz for that CPU. When the CPU wakes up again, the original tick frequency is restored [31]. This is called *tickless kernel mode* and yields energy savings of up to 70% [49]. However, altering the tick frequency requires writing to the `TSC_DEADLINE` MSR, inducing a VM exit.

All the above causes are correlated. When a thread blocks and there are no other runnable tasks for the vCPU, the guest usually disables its scheduler tick and halts it, resulting in 2 VM exits. When the thread is woken up again, 2 more VM exits follow for sending a `RESCHEDULE` IPI and reactivating the scheduler tick. Thus, each blocking operation results in up to 4 VM exits. Fig. 9 shows that each of these operations can be costly. We were especially surprised to find that `TSC_DEADLINE` MSR writes account for a $\delta\eta_r$ of up to 10%, since tickless kernels have been described before as having a positive effect on virtualization [50].

While 3 out of 4 VM exits associated to blocking synchronization are caused by MSR writes, fig. 9 shows that scheduling overhead still dominates. We found that scheduling is almost always triggered by a `HLT` VM exit. When halt polling is successful (i.e. the vCPU is woken up before the polling ends and is immediately rescheduled), the cost of handling this VM exit is limited. When it is unsuccessful however (i.e. the polling interval expires and the vCPU needs to be descheduled anyway), the cost becomes very high. Because cycles spent on unsuccessful polling only slow down the scheduling process, we consider them as scheduling overhead as well in fig. 9.

Halt polling has 3 interesting implications for virtualization overhead. Firstly, $\delta\eta_r$ is in general much higher for 32 vCPUs than for 64 in fig. 9. This is a consequence of the heuristics KVM uses to manage the polling threshold. If the poll was unsuccessful, KVM grows or shrinks the threshold if the vCPU was blocked for resp. a short or long time. [31]. As vCPU counts increase, so do contention, average blocking time and the polling threshold. At 64 vCPUs however, the average blocking time is so long that the polling threshold shrinks to 0. We confirmed this by measuring the success rate of halt polling, which drops from 30% on average for 4 vCPUs, to close to 0% for 64 vCPUs. Secondly, halt polling is largely responsible for the strange evolution of ω in fig. 8. By design, halt polling expends CPU cycles to improve performance, lowering ω ever more as the polling threshold grows up to 32 vCPUs. When the polling threshold shrinks back to 0 for 64 vCPUs, ω rises drastically as $\delta\eta_r$ drops at the expense of $\delta\eta_t$. Lastly, $\delta\eta_r$ is higher in the UC scenario compared to OC in fig. 9. Contrary to the UC scenario, polling can degrade system throughput in the OC scenario, as other VMs may use the cycles spent on polling to make progress. KVM solves this by disabling polling altogether when the CPU has other runnable tasks [31], reducing $\delta\eta_r$ in the OC scenario.

Halt polling overhead may vary between VMMs. In Xen HVM for example, halt polling is not implemented. $\delta\eta_r$ will thus be lower in the UC scenario for Xen than for KVM, while $\delta\eta_t$ will be higher. In the OC scenario, scheduling overhead for Xen will be comparable to KVM. On the other hand, as the root cause of the `TSC_DEADLINE` MSR writes lies in the guest OS, this overhead may vary between guests. The induced VM exits are handled comparably by Xen and KVM, as are IPIs. In terms of hardware, Intel and AMD offer unique APIC virtualization extensions (resp. APICv [16] and AVIC [22]). While implementation details differ, their effect and performance are similar. Both eliminate the need for VMM intervention to inject the IPI and acknowledge its receipt, but still require a VM exit to write the `ICR` MSR.

6.2.2 Virtual Memory Management

Fig. 9 shows that *Dedup* and *Vips* spend a lot of cycles on processing TLB shootdown IPIs. The TLB is a per-CPU cache that stores page table entries (PTEs) [16]. TLB consistency across different CPUs has to be maintained by the OS. When a process changes a PTE, the OS sends a *TLB shootdown* IPI to all other CPUs using the same virtual address space to make them flush the altered entry from their TLBs [31]. This induces a VM exit due to the `ICR` MSR being written.

The high-level cause of TLB shootdown IPIs is data sharing between threads. The exact amount of such IPIs is highly dependent on the application source code and underlying system libraries. The *glibc* memory allocator can in some cases perform excessive heap resizing operations that induce these IPIs. For example, when an application often allocates small amounts of memory at the top of the heap and frees it a short while later, glibc will trim the heap and return the pages to the OS, only to request them again soon after. This induces many `madvise` and `mprotect` system calls, which send the vast majority of TLB shootdown IPIs observed in fig. 9 [5]. While there are other causes of IPIs such as page migrations, we found them to be insignificant for our workloads compared to heap resizing.

The direct virtualization overhead for sending a TLB shootdown IPI is identical to that for rescheduling IPIs. As this overhead is handled comparably across hardware platforms and VMMs, similar performance can be expected for AMD- or Xen-based systems.

6.2.3 Spinning at Kernel Level

Some years ago, spinning at kernel level was a serious issue for overcommitted virtualized systems in the form of LHP and related issues, as described in §2.2. Our experiments however show that PLE is very effective at dealing with this. We found that for our experiments, only *Vips* in the OC scenario suffers from many PLE VM exits. While the overhead caused by these exits themselves is low, they invoke the scheduler, inducing significant scheduling overhead. As *Vips* incurs negligible `HLT` and preemption timer VM exits compared to the other workloads suffering high host-level virtualization overhead, almost all the scheduling overhead for *Vips* shown in fig. 9 can be attributed to PLE. Nevertheless, we consider this scheduling overhead acceptable, since it is comparable to that for other benchmarks in the OC scenario and the scheduler would otherwise be triggered anyway by other mechanisms.

Despite our reassuring results, PLE is not a fundamental solution, since it can only trigger a VM exit after some spinning has already occurred. Because this spinning takes place in the guest kernel, it is visible as guest-level overhead in fig. 4. Overall however, we are pleased to note considerable progress in dealing with LHP and related issues in recent years. Only half a decade ago, $\delta\eta_t$ was over 500% for *Dedup* on overcommitted systems, mainly due to LHP [8]. Thanks to PLE, $\delta\eta_t \approx -20\%$ (OC) or $\delta\eta_t \approx 50\%$ (OC_2).

AMD implements *pause filter (PF)* in its CPUs, which is identical to Intel's PLE [22]. Both solutions are treated equally by KVM as well. Xen source code reveals that it handles PLE/PF much like KVM. We thus conclude that spinning at kernel level has been tackled effectively across hardware and virtualization platforms.

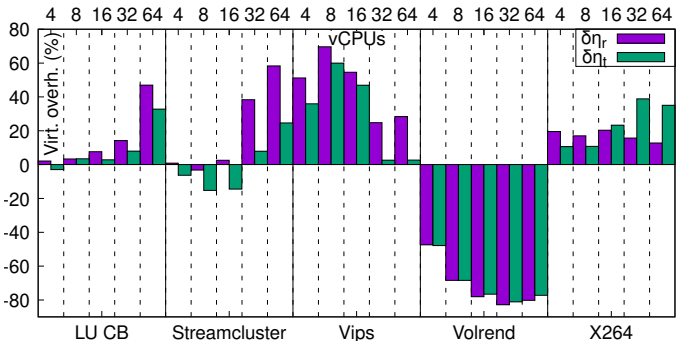


Fig. 10. Breakdown of the virtualization overhead in the OC2 scenario for the benchmarks that show high overhead in the OC scenario.

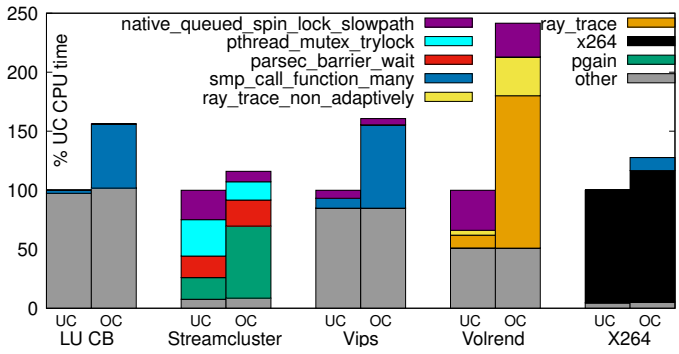


Fig. 11. Comparison of subroutine CPU profile between UC and OC virtualized execution with 64 vCPUs for the benchmarks of interest.

6.3 High Overcommitted Overhead

Naturally, we break down the overhead for this category in the OC scenario. We choose the OC_2 data set to eliminate the effects of server consolidation, as we are purely interested in the overhead itself. Fig. 10 shows the results.

The benchmarks in fig. 10 seem to consist of 2 subgroups: those with resp. positive (*LU CB*, *Vips*, *X264*) and negative (*Streamcluster*, *Volrend*) virtualization overhead. Note that besides overcommitting overhead, *Streamcluster* suffers from NUMA locality issues, distorting the results.

In an effort to understand the patterns in fig. 10, we compare the callstack of the UC and OC executions of the benchmarks in terms of CPU cycles. We only show the 64 vCPU variants, since fig. 10 indicates there is limited variance between vCPU counts. Fig. 11 shows the results.

Fig. 11 shows that for subgroup 1 in fig. 10 the system function `smp_call_function_many` is mainly responsible for the difference between UC and OC CPU time, while for subgroup 2 some application-level functions are the culprit. We discuss each group in detail below.

6.3.1 TLB Shootdown Preemption

`smp_call_function_many` is a system-level function used to send TLB shootdown IPIs. In the OC scenario, these IPIs thus become even more costly. Namely, the sending vCPU must synchronize with the receiving vCPUs by means of a spin lock before proceeding. When a receiving vCPU is not running, the sender must wait until it is rescheduled, which leads to excessive spinning. While PLE largely mitigates this, PLE itself is not cost-free (see §6.2.3). This also explains the many PLE VM exits for *Vips* observed in fig. 9. This is known as the *TLB shootdown preemption problem* [51].

6.3.2 User-Level Spinning

Streamcluster and *Volrend* show greatly increased CPU time for particular application functions in the OC scenario. By analyzing the source code of these functions, we found that they implement their own spinning-based synchronization at user level, rather than using kernel routines. Previous research has shown that many applications make use of similar primitives [52]. This leads to an LHP-like problem at user level. PLE can not intervene here, as it relies on the `PAUSE` instruction to work. User-level synchronization primitives rarely compile down to this instruction. Moreover, PLE only works in kernel mode (CPL=0) [16].

We use *Volrend* as an example to illustrate the user-level spinning problem, since it suffers the most from this issue. When analyzing the source code of the `Ray_Trace` function, which consumes approx. 10 times more cycles in the OC scenario in fig. 11, we find that it utilizes the following user-level spin-based barrier:

```
LOCK(Global->CountLock);
Global->Counter--;
UNLOCK(Global->CountLock);
while (Global->Counter);
```

Listing 1. User level spin-based barrier in *volrend*.

By definition, user-level spinning is heavily dependent on the application source code. Additionally, we found 2 factors that influence the severity of user-level spinning:

- Increasing thread- and vCPU counts lead to more intensive spinning synchronization, as shown in our experiments. This problem will thus gain importance towards the future, as VM sizes tend to grow.
- More frequent task switches increase the chance that a thread holding a lock gets preempted, increasing the severity of user-level spinning. Figures 4 and 10 prove this, as *Volrend* shows high overhead for the OC data set, but negative overhead for the OC_2 data set. Firstly this indicates that the overcommitted native execution is much slower than the undercommitted one, meaning that user-level spinning is also an issue when running natively. Secondly, the overcommitted virtualized execution is faster than its native counterpart because in each VM there is only one instance of the benchmark, while natively 2 instances are run within the same OS for the OC_2 data set. As time slices are allocated to vCPUs at a much larger granularity than to threads, it is much less likely that a lock-holding thread is preempted in a VM, thus reducing user-level spinning.

From fig. 4, it is clear that user-level spinning is an as of yet unaddressed issue with potentially severe performance implications in both native and virtualized contexts. On the other hand, fig. 11 shows a decrease in kernel-level spinning (`native_queued_spin_lock_slowpath`) and blocking synchronization (`pthread_mutex_trylock`) for *Streamcluster*, for reasons explained in §5.2. Combined with the NUMA issues for this benchmark identified above, this illustrates the complexity of quantifying overhead and categorizing the benchmarks.

Since user-level spinning originates from the application, it must be treated as a conceptual rather than an implementation issue from the VMM's perspective. Therefore, all VMMs and hardware are equally prone to this problem.

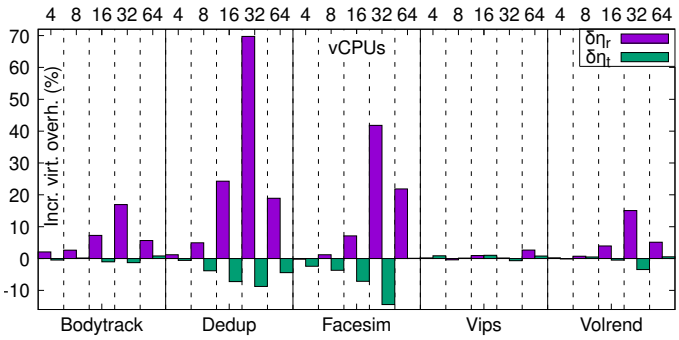


Fig. 12. $\delta\eta_r$ and $\delta\eta_t$ caused by halt polling for the benchmarks with high host overhead per vCPU count, UC.

7 Mitigation

It is evident that multithreading still induces substantial virtualization overhead stemming from various sources. Conceptually, we can group these sources in 2 categories: thread coordination and NUMA locality. The former can be split further into blocking synchronization, spinning synchronization and memory management. Below we highlight the challenges to further reducing overhead for each of these categories and discuss promising research directions.

7.1 Thread Coordination

7.1.1 Blocking Synchronization

Research efforts regarding overhead induced by blocking synchronization mainly focus on vCPU scheduling. Currently, halt polling is already adopted in KVM, albeit seemingly at a high cost (see §6.2.1). To clarify this perception, we compared $\delta\eta_r$ and $\delta\eta_t$ from the experiments in fig. 8 to identical experiments with halt polling disabled. Fig. 12 shows the results. We omit the OC scenario since the impact of halt polling is much smaller there, as discussed in §6.2.1.

Fig. 12 shows that halt polling is not very efficient. While it reduces $\delta\eta_t$ by up to 14%, this comes at a great cost in cycles. When performance is the only concern, this is justifiable. However, in these days of efficiency being a primary concern and cloud providers charging users by the CPU-ms, such situations are becoming a rarity. Moreover, halt polling is much less effective when the system is overcommitted and/or VM sizes are large, indicating that it is not a durable solution in heavily consolidated cloud environments, especially towards the future.

The above issues are inherent to the polling concept. It is very hard to balance performance and efficiency, especially on the overcommitted systems where any cycles spent on polling reduce throughput. The reluctance of Xen to adopt halt polling underpins this. Therefore more intelligent solutions are highly desirable. Existing research has attempted to replace polling by computation migrated from other vCPUs, but this introduces vCPU overloading as a side-effect [15]. A recent solution, vScale [53], can reduce such side-effects, but requires substantial changes to the guest OS.

IPI-induced overhead has received much attention from hardware manufacturers. APICv and AVIC reduce IPI-induced overhead by 60%. Nevertheless, our results indicate that this issue is still significant, especially since IPIs are often on the critical path. Strict co-scheduling could solve this problem since it would eliminate the need for intercepting IPIs because whenever a guest CPU sends an IPI, the receiving vCPU is guaranteed to be running on the intended pCPU. However, co-scheduling has its own issues such as CPU fragmentation [44].

Overhead related to management of the scheduler tick can be swayed by tweaking the boot parameter `CONFIG_NO_HZ` in Linux [49]. One can choose to never disable the scheduler tick, only disable it on idling CPUs (default), or disabling it on CPUs that have at most 1 runnable task. We found that never disabling the tick does not improve performance accordingly. While we did find a large reduction in writes to the `TSC_DEADLINE` MSR, the writes to the `ICR` MSR increased drastically, since the scheduler will be much more likely to send `RESCHEDULE` IPIs to idling cores when the tick was not disabled. Moreover, on highly overcommitted systems this may lead to overwhelming overhead, as the host must handle each guest's tick interrupts individually [50]. Disabling the tick for CPUs that have at most one runnable task can be a solution for some workloads, but for others such as *dedup* that have many more threads than cores by design this merely offsets the problem from the transition between 1 and 0 runnable tasks to that between 2 and 1.

All the issues with blocking synchronization are caused by discontinuous CPU availability to idle vCPUs. Efficiently increasing CPU availability in system software must handle difficult trade-offs (e.g., polling vs. blocking). System software has no direct knowledge of the workload, and must rely on heuristical approaches. Hardware solutions increase the hardware contexts in a core to guarantee a hardware context for each vCPU and ensure CPU availability without reducing throughput. As full-fledged hardware solutions are not readily available, enabling SMT can increase CPU availability and serve as a mitigation.

Because system solutions are challenging, an alternative approach is desirable. Literature shows that one such approach could be adopting a different application architecture that focuses on data parallelism and eliminating dependencies between threads as much as possible. Such application-level solutions can be highly effective for synchronization-heavy workloads [5]. The obvious downside of this approach is that it can be labor-intensive. However, solutions aiding in this process exist. For example, parallel patterns can abstract the implementation details of multithreading from developers. The authors of [54] have applied this technique to the PARSEC benchmark suite. We profiled their implementation to asses its effectiveness in reducing virtualization-sensitive synchronization operations. Fig. 13 shows the results for all the PARSEC benchmarks identified in §6.2.1 as having high synchronization overhead, broken down per vCPU count in the UC scenario. Comparable results are expected in the OC scenario.

Fig. 13 shows promising results. All synchronization operations have been reduced by up to 70%. The improvement tends to increase with vCPU count. One exception seems to be the `HLT` operations for *Dedup*. However, profiling *Dedup* in detail reveals that these operations are induced by I/O rather than synchronization. Fig. 13 also suggests this, as the `RESCHEDULE` IPIs are drastically reduced. This suggests that application-level solutions indeed have great potential.

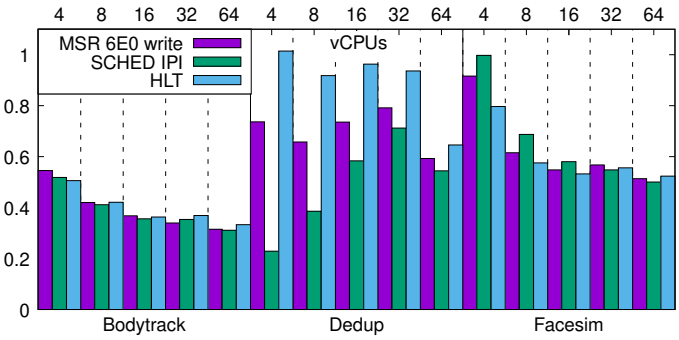


Fig. 13. Number of virtualization-sensitive synchronization operations for the P3ARSEC workloads relative to their original equivalents that show many such operations per vCPU count, UC.

Our results in §6.2.3 show that spinning in guest OS kernels is no longer a major concern for virtualized multithreaded applications, largely thanks to PLE. In Linux, this hardware mechanism is supplemented by PV-spinlocks, further reducing LHP-like problems in many critical areas [31].

On the other hand, user-level spinning remains a challenge. Apart from co-scheduling with its known issues, addressing this issue at system level is very challenging, because programs may implement spinning in many different ways. This makes detecting user-level spinning at runtime without knowledge of application specifics prohibitively expensive if not impossible. Because the problem originates in user space, solutions at application level are naturally highly appealing as such an approach allows tackling the issue directly and precisely. Below we discuss a few options:

• Alternative synchronization primitives could be used in a variety of ways. Firstly, spinning can manually be replaced by blocking synchronization in application source code. We explored this approach for *volrend*, finding that $\delta\eta_r$ and $\delta\eta_t$ were reduced by resp. 60% and 25% in the OC, 64 vCPU scenario. These results are somewhat modest since blocking synchronization is much more costly than spinning and introduces new host-level overhead (see §6.2.1). Alternatively, spin-then-block primitives which first spin for a short, heuristically determined time, after which they block can be used. Programming language APIs should provide such abstractions. Complementarily, compilers and interpreters could also be extended to detect basic user-level spinning constructs and replace them with these spin-then-block primitives.
• A solution exploiting PLE is possible by making user-level spin locks compile down to the `PAUSE` instruction, either through automated detection by the compiler/interpreter or through the use of dedicated primitives and extending the hardware so that PLE works at user level.
• A VMM-level solution could be to use pause exiting rather than pause-loop exiting, which generates a VM exit on each `PAUSE` instruction [16]. On such an exit, the VMM can reschedule the vCPU immediately for a while until a threshold is reached. If the vCPU keeps exiting, LHP is likely and another vCPU can be scheduled. This principle is similar to halt polling. While this approach may be more demanding in terms of resources, it does not burden application developers and does not require hardware extensions.

7.1.2 Memory Management

The last sizeable remaining challenge regarding virtualizing memory in x86 is TLB consistency. The core issue here is the fact that in x86, the TLBs are populated by hardware but -in contrast to other caches- have to be synchronized by the OS. Because of this, the contents of each TLB are opaque to the OS, leaving little room for optimization. Even in a native context, this can have problematic performance implications [55]. In VMs, TLB shootdown cost is even greater due to the ICR MSR write VM exits associated to sending IPIs and TLB shootdown preemption. Our results from §6.2.2 show that despite recent hardware improvements (APICv/AVIC), TLB shootdown overhead can still be significant. We envision several possible mitigation strategies:

• Many alternative TLB designs have been proposed, e.g. using a shared TLB or implementing various forms of TLB synchronization in hardware [56]. These proposals could easily be extended to work for virtualized systems since modern TLBs contain a VM ID tag for each TLB entry [16].
• Strict co-scheduling could eliminate the need for VMM handling of ICR writes as well as TLB shootdown preemption. Implementation and drawbacks are already discussed in §7.1.1 in the context of scheduling IPIs.
• Since the root cause of most TLB shootdowns for computation-intensive multithreaded workloads lies at application level, altering source code is a viable option as well. However, besides laborious, this is highly challenging since memory allocators are very complex. Identification and correction of problematic code without greatly compromising memory efficiency is therefore not an easy task.
• Application memory allocators could be tweaked so that they call system routines inducing IPIs much less often at the inevitable expense of some memory efficiency. Essentially, we argue that rising TLB shootdown cost due to increasingly parallel and virtualized systems warrants reconsidering the trade-off between memory efficiency and performance from a memory allocator design perspective. We consider this the most promising approach.

7.2 NUMA Locality

Several approaches already exist to deal with excessive memory latency in VMs. Two methods are common: passing through the hardware NUMA architecture to the VM, and using automated heuristical NUMA-optimization algorithms at VMM level. The former method is available in every major VMM [4]. The latter is integrated in most common VMM and OS schedulers. Besides, one can use dedicated daemons that advise the kernel on optimal memory locality. Optimizing these automated techniques is the subject of active research, independent of the VMM used [4], [47].

We assess the effectiveness of both NUMA-optimization methods described above for the 64 vCPU, UC variant of the benchmarks from figure 6. Since the former method relies on the guest OS scheduler, identical performance can be expected for any VMM. For the latter method, much more variance between VMMs is possible, since each VMM uses its own algorithm. However, results for 1 algorithm may give some indication of how others will behave. We pick `numad`[3], which is a common dedicated NUMA local-

3. https://linux.die.net/man/8/numad

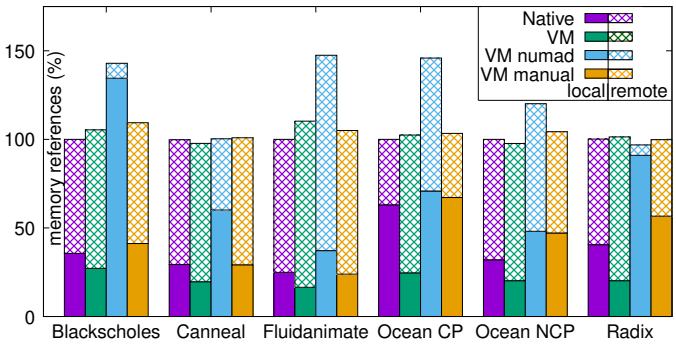


Fig. 14. Memory locality normalized to native for varying NUMA management techniques applied to the benchmarks with high guest overhead in the UC, 64 vCPU scenario.

ity management daemon for Linux/KVM. We analyze the number of local and remote memory accesses using the tool `pcm-numa`[4]. Fig. 14 shows the results, normalized to native.

As fig. 14 shows, memory locality is greatly reduced for all benchmarks when run in a VM without optimizations. Manual NUMA exposure mitigates this issue entirely. This solution however reduces the potential for resource consolidation, since vCPUs can no longer be migrated between sockets without compromising the advantages of passing through the NUMA architecture to the guest. While `numad` achieves even better locality than manual passthrough, its performance is unpredictable. After analyzing $\delta\eta_r$ for the benchmarks from fig. 14, we found that for *Blackscholes*, *Canneal* and *Radix*, using `numad` reduces $\delta\eta_r$, despite often increasing memory accesses. For the other benchmarks however, an additional $\delta\eta_r$ of up to 45% is observed.

From fig. 14 it is unclear which NUMA-optimization method is preferable, since their relative performance varies greatly between benchmarks. We investigate this further by analyzing the benchmark for which `numad` shows the worst $\delta\eta_r$, *Ocean CP*, in detail. Firstly, we found that `numad` itself consumes many cycles. Secondly, *Ocean CP* is bottlenecked by memory bandwidth, as noted in §6.1. `Numad` seems to ignore this metric, only optimizing locality. Because more threads are scheduled on the same socket, the bottleneck magnifies. On the other hand, manual exposure is tedious and limits flexibility regarding resource allocation and VM migration. Thus, neither solution is universally superior.

We were surprised to find that NUMA-locality is still such a severe issue in virtualized systems. The underlying issue is dynamic scheduling of vCPUs. If we can guarantee that a vCPU will always be scheduled on the same NUMA node, the NUMA architecture can be automatically exposed to the VM. The strict co-scheduling and SMT-based approaches proposed in §7.1 could provide exactly this guarantee. Alternatively, if improvements are made to `numad` to take more metrics into account, it can be highly effective as well. Lastly, [4] proposes extended paravirtualization (XPV) (applied in Xen). This method alters the guest OS so that it can dynamically change its NUMA configuration through communication with the hypervisor. While native performance can be achieved like this without the flexibility restrictions of classic NUMA passthrough, it requires changes to the guest kernel, limiting its potential.

4. https://github.com/opcm/pcm

8 Threats to Validity

As this work is based on controlled experiments, it is empirical in nature. Threats to validity are inherent to any such endeavors. We aim to provide the reader with the correct context in which to interpret our results by discussing the main threats to validity for this study below:

- Firstly, any work measuring benchmark performance is faced with non-determinism inherent to some system components (e.g. variations in scheduling, external interrupts,...). Despite our best efforts as described in §4, we found that a variance of approx. 5% is to be expected in all measurement results. Particularly benchmarks that suffer from NUMA locality issues are sensitive to such non-deterministic performance fluctuations, since slight variations in scheduling heavily influence their overhead.
- Naturally, our measurement results are only valid for our exact system configuration. Nevertheless, the identified high-level overhead causes are conceptual in nature, regardless of system or workload specifics. Moreover, in §6 we reasoned about how our findings would translate to other platforms. In this way, this paper is to a large extent implementation-agnostic, despite its empirical nature.
- While we are confident that we identified the vast majority of remaining challenges within the scope of the paper, it is impossible to guarantee this. Due to the many layers of abstraction in virtualized systems and quasi endless variety of workloads these systems may be tasked with, some issues that did not emerge in our analysis might surface under very specific circumstances.
- Software typically evolves rapidly. While we used the newest version of Ubuntu when we started this study, by the time we completed it many newer Linux/Ubuntu versions have been released. While it is impossible to redo our entire analysis every time a new kernel version is released, we provide a strong indication that our results will be valid for a long time to come by running the 64 vCPU variant of 1 benchmark from each category defined in §6 using the latest stable Linux kernel on the host at the time of finishing this project: 4.19.88. We chose the benchmarks *Bodytrack*, *Ferret*, *Ocean CP* and *X264* for this purpose.

We found that all tested benchmarks yield almost identical results on the new kernel compared to kernel 4.15, with the exception of *X264* in the overcommitted scenario. We found that the overhead induced by TLB shootdown preemption has disappeared. After analyzing the new kernel's source code, we found that in kernel 4.16 a patch was implemented that mitigates this problem entirely by paravirtualizing TLB shootdowns in Linux for KVM [57]. The guest only sends IPIs to vCPUs that are running. Other vCPUs flush their TLB on re-scheduleding. Very recently, a similar solution has been implemented for Xen [58].

Overall, we conclude that while some variance in the exact results is to be expected, our findings are solid and largely independent of variations in system settings or non-deterministic factors. Since our test system sports all contemporary industry-standard enhancements to mitigate virtualization overhead, practitioners are likely to experience performance close to our test results for several more years as any research advancements only slowly trickle down into industry due to reliability and compatibility concerns.

9 Related Work

While virtualization overhead is a popular research topic, most studies fail to provide deep insight into overhead causes or their link to system and application effects; let alone differentiate between the latter [2], [59], [60]. More profound work tends to have a very narrow scope, e.g. nested paging [61], NUMA locality [47] or I/O [62]. Nevertheless, these studies have pinpointed various major causes of virtualization overhead, such as false cache sharing, extra iTLB misses and poor I/O performance, resulting in various hardware improvements being implemented [11], [63].

In the context of multithreading, existing studies have identified two main drivers of virtualization overhead: interaction between threads which often requires costly traps to the VMM, e.g. for handling IPIs [5], [8], [15], [44], [51] and the semantic gap between the VMM and guest OS, which results in a variety of issues for particularly spinning synchronization (e.g. LHP) [14], [64]. Due to huge advancements in hardware and VMM design in recent years such as PLE and halt polling however, many of these studies have become inaccurate to the point of obsolescence.

Concerning the overcommitted scenario, literature has shown performance isolation and fairness issues caused by resource contention. Due to poor management of shared resources such as cache space, memory bandwidth and CPU time by the VMM, some virtualized applications may be unfairly deprived of resources in favor of competing workloads on the host [65], [66]. Various techniques such as cache space partitioning and improved scheduling algorithms have been developed to address this. These fairness issues are not in the scope of our paper, because they are not specific to virtualization, nor multithreaded applications.

10 Conclusion

Thanks to persistent efforts from academia and industry, hardware-assisted x86 virtualization induces minimal overhead for sequential computation-intensive workloads on modern platforms. Unfortunately, this is not yet the case for their multithreaded counterparts. Overhead may manifest itself in many different ways. The perceived application effects may differ greatly from the underlying impact on the system. Both of these may vary greatly between workloads and system configurations.

The main causes of the overhead are thread-coordination and NUMA management. Ongoing efforts on these fronts prove that both these issues are challenging to deal with at system level. We propose that increased attention be given to application-level solutions, especially since our first exploratory steps in this direction yield promising results.

While this paper touches on many known issues - notwithstanding some novel findings-, we are the first to perform a broad systematic analysis of virtualization overhead related to multithreading on modern systems. In this way, we have provided a clear overview of the state of the art, remaining challenges and the link between overhead causes and effects. Especially considering the enormous advances in virtualization technology in the last decade which render most established related work obsolete, we consider this study a valuable asset to the research and system development communities alike.

Acknowledgments

We thank the reviewers and the editor for their constructive comments. This work is funded in part by the US National Science Foundation grant CCF 1617749, the Flemish FWO grant V433819N and KU Leuven JUMO grant 19005.

References

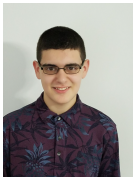
[1] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," in *ISCA'14*. IEEE, 2014, pp. 325–336.
[2] J. Han, J. Ahn, C. Kim, Y. Kwon, Y.-r. Choi, and J. Huh, "The effect of multi-core on hpc applications in virtualized systems," in *Euro-Par 2010 Parallel Processing Workshops*. Springer, 2011, pp. 615–623.
[3] J. Fisher-Ogden, "Hardware support for efficient virtualization," *University of California, San Diego, Tech. Rep*, 2006.
[4] B. Bui, D. Mvondo, B. Teabe, K. Jiokeng, L. Wapet, A. Tchana, G. Thomas, D. Hagimont, G. Muller, and N. Depalma, "When extended para-virtualization (XPV) meets NUMA," in *EuroSys'19*, 2019, pp. 1–15.
[5] S. Schildermans and K. Aerts, "Towards high-level software approaches to reduce virtualization overhead for parallel applications," 12 2018, pp. 193–197.
[6] J. T. Lim and J. Nieh, "Optimizing nested virtualization performance using direct virtual hardware," in *ASPLOS '20*, 2020.
[7] L. Youseff, K. Seymour, H. You, D. Zagorodnov, J. Dongarra, and R. Wolski, "Paravirtualization effect on single-and multi-threaded memory-intensive linear algebra software," *Cluster Computing*, vol. 12, no. 2, pp. 101–122, 2009.
[8] X. Ding and J. Shan, "Diagnosing virtualization overhead for multi-threaded computation on multicore platforms," in *CloudCom'15*, 2015, pp. 226–233.
[9] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 610–614.
[10] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in *VEE'05*.
[11] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *HPCA'10*, 2010.
[12] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," in *IPDPS 2010*. IEEE, 2010, pp. 1–12.
[13] T. Friebel and S. Biemueller, "How to deal with lock holder preemption," *Xen Summit North America*, 2008.
[14] J. Ouyang and J. R. Lange, "Preemptable ticket spinlocks: Improving consolidated performance in the cloud," in *VEE'13*, 2013.
[15] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, "Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications," in *USENIX ATC 2014*, 2014, pp. 73–84.
[16] Intel Corporation, "Intel 64 and ia-32 architectures software developer's manual - volume 3," September 2016. [Online]. Available: https://intel.ly/3geYN2Z
[17] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.
[18] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.
[19] T. L. Foundation, "Home - xen project." [Online]. Available: https://xenproject.org/
[20] "Server virtualization software: vsphere," Dec 2019. [Online]. Available: https://www.vmware.com/products/vsphere.html
[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.
[22] AMD, "AMD64 architecture programmer's manual: Volumes 1-5," 11 2020. [Online]. Available: https://www.amd.com/system/files/TechDocs/40332.pdf
[23] M. Zabaljauregui, "Hardware assisted virtualization intel virtualization technology," *accessed at linux. linti. unlp. edu. ar/images/f/f1/Vtx. pdf*, pp. 1–54, 2008.
[24] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *OSDI'08*, 2008.

[25] "Intel resource director technology (Intel RDT)." [Online]. Available: https://intel.ly/2YetmQ0
[26] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ASPLOS'06*, 2006.
[27] "Virtio: Paravirtualized drivers for KVM/linux," 2019. [Online]. Available: https://www.linux-kvm.org/page/Virtio
[28] S. W. Devine, L. S. Rogel, P. P. Bungale *et al.*, "Virtualization with shadow page tables," Jun. 11 2013, uS Patent 8,464,022.
[29] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 26–35, 2008.
[30] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont, "The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock)," in *EuroSys '17*, 2017, pp. 286–297.
[31] Torvalds, "torvalds/linux," Jun 2019. [Online]. Available: https://github.com/torvalds/linux
[32] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu, "Optimizing virtual machine scheduling in NUMA multicore systems," in *HPCA'13*, 2013, pp. 306–317.
[33] "Domain xml format." [Online]. Available: https://libvirt.org/formatdomain.html
[34] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen, "vCPU as a container: Towards accurate CPU allocation for VMs," in *VEE'19*, 2019.
[35] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *ICDCSW'17*, 2017.
[36] AMD. Leadership high performance computing. AMD. [Online]. Available: https://bit.ly/3aCEZWa
[37] R. Lingeswaran. (2017, 11) Cloud war – openstack vs vmware vsphere vs amazon aws. [Online]. Available: https://bit.ly/3hgcsrK
[38] G. P. C. Tran, Y.-A. Chen, D.-I. Kang, J. P. Walters, and S. P. Crago, "Hypervisor performance analysis for real-time workloads," in *HPEC'16*, 2016.
[39] The Red Hat Enterprise Linux Team. (2019, 12) Red hat: Leading the enterprise linux server market. [Online]. Available: https://red.ht/3aCFGPg
[40] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *EuroSys 2011*. ACM, 2011, pp. 257–272.
[41] VMware, "Host power management in VMware vSphere 5.5." [Online]. Available: https://bit.ly/3hdMBRo
[42] Distrowatch. (2020) Distrowatch project ranking. [Online]. Available: https://bit.ly/3hhY3eM
[43] X. Zhan, Y. Bao, C. Bienia, and K. Li, "PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X," *SIGARCH Comput. Archit. News*, vol. 44, no. 5, p. 1–16, 2017.
[44] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs," in *ASPLOS'13*, 2013.
[45] C.-Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *ICDCS'88*, 1988.
[46] U. Drepper. (2007, 11) Memory part 7: Memory performance tools. [Online]. Available: https://lwn.net/Articles/257209/
[47] G. Voron, G. Thomas, V. Quema, and P. Sens, "An interface to implement NUMA policies in the xen hypervisor," in *EuroSys'17*, 2017, pp. 453–467.
[48] KVM Developers, "The kvm halt polling system." [Online]. Available: https://bit.ly/2Q63Ckn
[49] "NO_HZ: Reducing scheduling-clock ticks." [Online]. Available: https://bit.ly/3gceTdQ
[50] S. Siddha, V. Pallipadi, and A. Ven, "Getting maximum mileage out of tickless," in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 201–207.
[51] J. Ouyang, J. R. Lange, and H. Zheng, "Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs," *VEE'16*, 2016.
[52] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad hoc synchronization considered harmful," in *OSDI'10*, pp. 163–176.
[53] L. Cheng, J. Rao, and F. C. M. Lau, "vscale: Automatic and efficient processor scaling for smp virtual machines," in *EuroSys '16*, 2016.
[54] M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, and M. Torquati, "P3arsec: towards parallel patterns benchmarking," in *Proceedings of the Symposium on Applied Computing*. ACM, 2017, pp. 1582–1589.
[55] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory," in *PACT 2011*, pp. 340–349.
[56] S. Mittal, "A survey of techniques for architecting TLBs," *Concurrency and computation: practice and experience*, vol. 29, no. 10, 2017.
[57] W. Li, "Kvm: X86: Add paravirt tlb shootdown," Nov 2017. [Online]. Available: https://lwn.net/Articles/740363/
[58] P. Monne, Roger. (2020, 1) [v2,3/3] x86/tlb: use xen l0 assisted tlb flush when available. [Online]. Available: https://patchwork.kernel.org/patch/11327803/
[59] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu, "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks," in *IEEE BigData Congress'13*, 2013.
[60] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments," in *IEEE CLOUD'11*, 2011.
[61] T. Merrifield and H. R. Taheri, "Performance implications of extended page tables on virtualized x86 processors," ser. VEE'16, 2016.
[62] J. Li, S. Xue, W. Zhang, Z. Qi *et al.*, "When I/O interrupt becomes system bottleneck: Efficiency and scalability enhancement for SR-IOV network virtualization," *IEEE TCC*, vol. 7, no. 4, 2019.
[63] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization." *Intel Technology Journal*, vol. 10, no. 3, 2006.
[64] S. Kashyap, C. Min, and T. Kim, "Scaling guest OS critical sections with eCS." USENIX ATC 18, 2018.
[65] Y. Zhao, J. Rao, and Q. Yi, "Characterizing and optimizing the performance of multithreaded programs under interference," in *PACT 2016*, Sep. 2016, pp. 287–297.
[66] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *ISPASS'07*, 2007, pp. 200–209.

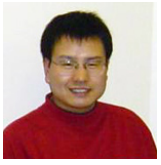
Stijn Schildermans is a PhD candidate at KU Leuven. He earned his Master's degree in Electronics & ICT Engineering Technology in 2017 from UHasselt and KU Leuven. His research interests include virtualization, cloud computing and functional programming. For his PhD project, he aims to reduce virtualization overhead from a high-level software perspective.

Jianchen Shan received the PhD degree in computer science from New Jersey Institute of Technology. He is an assistant professor with Hofstra University. His research interests include operating system, virtualization, cloud computing and high performance computing.

Kris Aerts is an Associate Professor and senior lecturer at KU Leuven, campus Diepenbeek. He received a PhD in computer science from KU Leuven. His research focusses mainly on practical applications of functional programming with industrial collaboration in mind. In recent years he started focusing on cloud computing as well.



Jason Jackrel is an undergraduate student in the Fred DeMatteis School of Engineering and Applied Sciences at Hofstra University. He will graduate with a BS in Computer Engineering in 2021. His research interests include Virtualization and Low-level hardware/software analysis.



Xiaoning Ding is an Associate Professor at New Jersey Institute of Technology. His interests are in the area of experimental computer systems, such as distributed systems, virtualization, operating systems, and storage systems. He earned his Ph.D. degree in computer science and engineering from the Ohio State University.