Maximizing VMs' IO Performance on Overcommitted CPUs with Fairness

Tong Xing[*]
The University of Edinburgh
tong.xing@ed.ac.uk

Cong Xiong
The University of Edinburgh
c.xiong-1@sms.ed.ac.uk

Chuan Ye
Huawei Cloud
yechuan@huawei.com

Qi Wei
Huawei Cloud
weiqi4@huawei.com

Javier Picorel
Huawei Cloud
javier.picorel@huawei.com

Antonio Barbalace[†]
The University of Edinburgh
antonio.barbalace@ed.ac.uk

Abstract

To improve resource utilization and reduce costs many Cloud providers adopt virtual machines (VMs) overcommitment. While effective, this strategy may lead to adverse outcomes, significantly affecting a VM IO performance when one virtual CPU (vCPU) is preempted by another vCPU within the same runqueue of the VM scheduler – i.e., same physical CPU (pCPU). Additionally, the responsiveness of a VM is reduced during the inactive time of the vCPU, and it necessitates an extra schedule timeslice to react to any IO event. While such problems have been studied in academia and industry, no previous solution has been deployed in production. This is because for example certain solutions require modifications of the guest VM, which is in contrast with industry requirements.

We propose Anubis, a new IO-aware VM scheduler targeting Linux KVM, the most popular VMM in today's Clouds, without requiring any guest VM modifications. Anubis shortens the IO event pending time by lightweight monitoring IO events including interrupt delivery and KVM exit. For the vCPU running the IO activity, Anubis provides an accurate boost, which is exclusively active only during the period when the vCPU has IO activity. While the IO performance is maximized, Anubis still guarantees fairness among VMs. The vCPU that doesn't have IO activity and belongs to the same VM will voluntarily yield the computing resources to counterbalance the unfairness created by the vCPU that has been given a performance boost. Overall, Anubis is a practical solution that provides close-to-non-overcommit performance for IO workloads in VM overcommitted scenarios.

[*]Tong Xing has started this work when at Stevens Institute of Technology, Hoboken, NJ, USA

[†]Antonio Barbalace has started this work when at Huawei Technology, Munich, Bavaria, Germany

CCS Concepts

• **Software and its engineering** → **Scheduling**; **Cloud computing**; **Virtual machines**.

Keywords

Overcommit, compute resources, virtualization, fair scheduling, Linux, KVM, low-latency, IO performance

ACM Reference Format:
Tong Xing, Cong Xiong, Chuan Ye, Qi Wei, Javier Picorel, and Antonio Barbalace. 2023. Maximizing VMs' IO Performance on Overcommitted CPUs with Fairness. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624649

1 Introduction

It is well known that many of the workloads running in the Cloud are not always busy [19]. Hence, Cloud providers consolidate multiple workloads together, on a single physical server, overcommitting servers' hardware resources – starting from a per-workload declared resources demand, usually estimated as the worst case/peak hardware resources required [4, 16, 18, 48]. This paper focuses on *CPU resources*, and looks at *multiple-CPU Virtual Machines* (VMs) as single-tenant workload bearers. When consolidating several VMs on the same physical server, while overcommitting resources, the virtual CPUs (vCPUs) of different VMs are going to potentially run on the same physical CPU (pCPU). The hypervisor (or Virtual Machine Monitor, VMM) scheduler time-multiplexes different vCPUs on each pCPU, trying either to meet a specific SLA, or to achieve the fair sharing of a pCPU by different vCPUs, while balancing the load among pCPUs [12]. Herein, we focus on the latter – the *fair scheduling* [12, 37] of *overcommitted VMs*, which unfortunately, due to the semantic gap

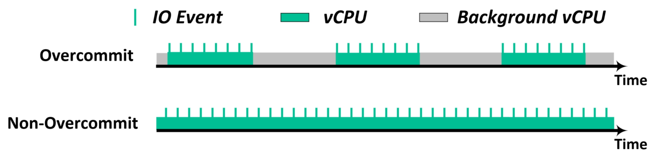


Figure 1: ***VM overcommitted impact on IO Event***

between the hypervisor scheduler and the software running in the VM, often creates VM IO performance degradation, especially for low-latency IO workloads.

The Problem. Because by overcommitting resources several vCPUs are time multiplexed on a single pCPU, a single vCPU doesn't run all the times. The tasks of a vCPU run only when that vCPU is running: when a vCPU exhausts its time slice, it is preempted, and paused – i.e., inactive, causing also its tasks to pause execution. **Fig. 1** compares the case of overcommit (2 vCPUs on 1 pCPU), with non-overcommit: IO events are not delivered when a vCPU is inactive. This can lead to several issues, particularly for IO-related workloads: (1) as the vCPU is paused, all interrupts, including the ones from IO devices and inter-processor-interrupts (IPIs), cannot be processed timely. Consequently, IO-related tasks cannot respond quickly to incoming interrupts; (2) the vCPU could be preempted in the middle of IO task processing, leading to substantial tail latency for the IO-related service; (3) the throughput of IO-related tasks could be impacted because fewer IO requests are processed per unit of time, this results in fewer responses being issued, which in turn triggers fewer requests in the subsequent service round. *How to alleviate the impact of vCPUs becoming inactive?* To put it in another way, could the performance of VMs' IO tasks in an overcommitted scenario be as good as their performance in a non-overcommit scenario?

Current Solutions. The problem of *IO performance degradation due to inactive vCPUs in an overcommitted scenario* is a classical problem in the field of systems research [15, 20, 21, 23, 34–36, 40, 44, 45, 52, 62, 63, 65, 70, 71]. Existing most related works mainly adopt one of two strategies: (1) patching the VM scheduler to minimize vCPU inactivity periods (vSlicer [70], vBalancer [15], AQL_sched [65]); (2) tracking IO tasks execution within vCPUs to guarantee they can process IO events without interruption – i.e., avoid descheduling (xBalloon [63], vMigrater [35], partial-boost [40]).

Those approaches have different limitations detailed in **§ 3**. For example, the state-of-the-art work on the topic – vMigrater, requires modifications to the guest VM – i.e., it is not transparent/no legacy support, which compromises its direct applicability in Cloud environments. Additionally, vMigrater demonstrated to be effective for VMs with a large number of vCPUs, while less effective on VMs with a few vCPUs (e.g., 1, 2, 4). However, it is known that more than 86% of the VMs in the Cloud use 4 vCPUs or less [26] – hence, vMigrater is not a generic solution.

Our approach draws inspiration from these previous works, but doesn't merely combine them. Instead of simply reducing vCPU inactivity time, we propose enhancing the vCPU's responsiveness by prioritizing VMs that receive IO event. Moreover, we suggest a shift in focus from ensuring the priority of the IO task inside the vCPU to the priority of the vCPU in which the IO task(s) is(are) running.

Anubis. We observe that if an application doing IO would have run directly on an OS sitting on bare-metal, instead than in a VM, the OS scheduler would have wake up the application waiting for IO soon after the IO interrupt is received. Hence, we asked ourselves: *can the hypervisor identify the IO events occurring in a VM and change VM's vCPUs scheduling decisions accordingly?* For example, by immediately waking up a non-running vCPU when it receives an IO interrupt to enable low-latency IO processing.

To optimize IO performance, it is crucial to give priority to the vCPU running IO task processing, ensuring that it will not be preempted. Thus, *can we accurately identify the start and end point of an IO event within a vCPU?* While this sounds promising from an IO performance point of view, the idea may lead to unfairness with the co-executing VMs, which may suffer from the reduced amount of resources. Hence, *can the new scheduler conserve fairness?*

We designed and implemented *Anubis* to answer the above questions. Anubis is a new IO-aware VM scheduler that extends widely-used traditional fair scheduling algorithms [12] implemented in modern Operating Systems (OSes) and hypervisors – hence, it strives to make fair scheduling decisions for the VMs. To reduce the vCPU inactivity Anubis carefully traces the interrupt delivery from the hypervisor and ensures the interrupt will be processed once it is delivered, while this may increase the number of context switches, we evaluate its overhead, which is minimal. To maintain the priority of the vCPU that is running the IO, Anubis uses lightweight VM introspection techniques to track IO events per vCPU, requiring *no guest VM software modification* whatsoever. The accuracy of detecting IO events is important, Anubis boosts the priority of the relevant vCPUs only during the IO events period. To maintain fair VM scheduling decisions Anubis introduces a debt-like system.

Anubis is implemented around the Linux KVM hypervisor. Therefore, it extends the Linux kernel's CFS scheduler and the Linux's KVM subsystem. Anubis has been thoroughly evaluated: in all overcommit scenario(s) it improves the IO performance close to the non-overcommit scenario. Hence, significantly reducing IO applications' latency.

Contributions. We make the following key contributions:

- First, we identify how the fair scheduling of VMs is a major factor affecting the IO performance of overcommitted VMs. Moreover, we analyze why previous works failed in providing solutions that can be deployed by Cloud providers.

- Second, we introduce Anubis, which improves the IO performance while conserving fairness. Anubis doesn't require any guest VM modification, and it is designed around three key ideas: mitigate the impact of vCPU inactivity, accurately maximize the IO performance, and ensure overall fairness; each introducing new mechanisms and policies.
- Third, we implemented and exhaustively evaluated a prototype of Anubis based on (one of the latest) vanilla Linux KVM releases. Showing that Anubis is easy to deploy, and it actually improves the IO performance with fairness.

The remainder of this paper is organized as follows. **§ 2** introduces the background and motivation of Anubis. **§ 3** introduces the previous works. **§ 4** presents the design principles and details of the Anubis. **§ 5** describes implementation details. **§ 6** presents the evaluation of the Anubis, and **§ 7** concludes.

2 Background & Motivation

I/O performance problems brought up by vCPU inactivity have been studied before, but despite the many proposed solutions, none have been integrated into production due to fundamental problems. In fact, the work in [40] only targets single-CPU VMs, which are indeed very common, but multiple-CPU VMs are even more popular today [26]. At the same time, recent works targeting multiple-CPU VMs [35, 63] require guest VM modifications – including user and kernel software, which makes them uneasy to be implemented in the Cloud.

2.1 Hypervisor

Hypervisors, or Virtual Machine Monitors (VMMs), are software, firmware, or hardware that create and manage virtual machines. There are mainly two types of VMMs: *type 1*, like Xen [58], and *type 2*, like Linux KVM [41]. Type 1 VMMs run directly on the bare-metal – without any operating system or runtime in between. The scheduler is part of the VMM itself. For example, the Xen hypervisor implements its own scheduling algorithm(s), being *credit2* the default [69]. Type 2 VMMs run atop an operating systems, the operating system provides support for devices, memory management, etc. The operating system scheduler does schedule the VMs together with processes and threads – i.e., the VMM does not implement a scheduler. Hence, a Linux KVM virtual machine is likely scheduled by the Linux's fair scheduling algorithm (details in **§ 2.2**).

Similar to Linux KVM's fair scheduler, also Xen's credit2 tries to maintain fairness among running vCPUs. With both schedulers, a vCPU running an IO task usually has higher priority because it consumes less CPU time while waiting for IO events. In the case of the credit2 scheduler, a vCPU running an IO task would accrue credits, and credit2 prioritizes a vCPU with higher credits by letting it preempt other vCPUs. However, a vCPU that runs IO tasks as well as CPU-bound tasks will not be prioritized by credit2. This is because such vCPU consumes credits by running its own CPU-bound task instead of waiting. Further details in **§ 2.4**.

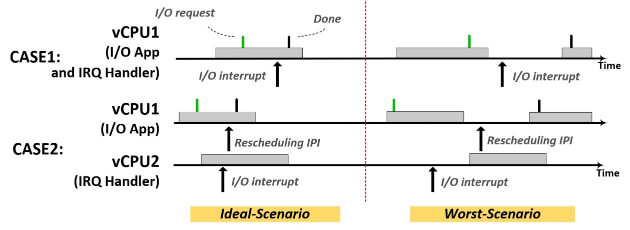


Figure 2: ***Ideal and worst scenarios of two cases in which vCPU scheduling may affect IO performance.***

Currently, Anubis targets type 2 VMMs because they are widespread [55, 73]. In fact, Linux KVM is getting more and more traction in the Cloud, and the most recent research in virtualization has shifted from Xen to Linux KVM, see **§ 3**. However, Anubis's idea can be applied to type 1 VMMs, and specifically to Xen by replacing KVM's `vruntime` (see below) to Xen's credit quantum.

2.2 Fair Scheduling

Linux by default is using the Completely Fair Scheduler (CFS) algorithm to schedule tasks (threads or processes). The primary goal of CFS is to ensure fair distribution of CPU time among threads, resulting in a balanced system. The key for determining if a thread should be scheduled to run is its "virtual runtime" (`vruntime`), which represents the amount of CPU time a process has already consumed. Briefly, the thread with the lowest `vruntime` always has the highest priority.

IO tasks often sleep while waiting for IO events, resulting in a smaller accumulated `vruntime`. This gives them a higher priority because CFS always selects the task with the lowest `vruntime`. As a result, when an IO task wakes up, it will preempt the running task, if any.

2.3 Interrupt Handling

The fair scheduling of different vCPUs equally splits the pCPU time among vCPUs, and traditionally the VMM scheduler doesn't know if a guest software is waiting for IO, or producing IO, etc. Hence, if an IO event for an application is received by the VMM just after the target VM's vCPU has been preempted in order to make another vCPU running, such event (likely an interrupt) will be delivered to the target VM's vCPU at its next timeslice, which can introduce a significant delay to the event processing, and degrade performance. This is depicted as **CASE 1** in **Fig. 2**. **CASE 2** in the same Figure shows a more unfortunate scenario where two timeslices need to elapse before an IO event is delivered to the application. When a VM with multiple vCPUs receives an interrupt on a vCPU that was not supposed to process it – because the application requesting the IO is on another vCPU, the receiving vCPU will generate a rescheduling IPI to the vCPU with the application. However, the rescheduling IPI maybe sent to a vCPU that has just been preempted – thus, a timeslice should pass before such IPI is processed. Moreover, the actual interrupt may be delivered to a just preempted vCPU, adding another timeslice to be waited for, in order to deliver the IO event to the application in the VM.

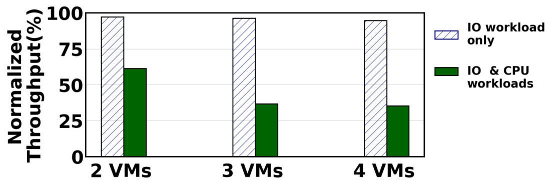


Figure 3: ***sysbench-seqrd [64] running without (IO Workload only) and with CPU-bound workload (IO & CPU workload) results normalized to the non-overcommitted case. All VMs with 4 vCPUs, overcommit ratios 2/3/4:1.***

2.4 Semantic Gap

In the overcommitted scenario, multiple vCPUs are sharing the same pCPU. Between such vCPUs, in Linux KVM the vCPU that has the lowest `vruntime` will have the highest priority among others. To the host CFS, a vCPU is just another task/thread, and because of the semantic gap, it doesn't know what applications are running inside the vCPU.

If a vCPU that is running an IO workload shares a pCPU with other vCPUs running a CPU-bound workload, the host CFS handles this case naturally well maintaining *the same IO performance as the non-overcommit case.* Refer to **IO workload only** bars in **Fig. 3**. Because the IO thread will yield the CPU when waiting for an IO event, the vCPU containing it will also yield the pCPU – going into idle, because there is nothing to execute. As a result of that, a lower `vruntime` is accumulated. When an IO event for such IO thread is received, the host CFS will schedule the vCPU containing it to run immediately.

However, if a vCPU contains both IO and CPU-bound workloads, the behaviour changes, see **IO & CPU workloads** bars in **Fig. 3**. Because the CPU-bound workload would continue to execute when the IO workload is waiting, the vCPU will not yield the pCPU, and its `vruntime` will accrue – like CPU-bound only vCPUs. This will deprioritize the vCPU with mixed workload that will not be able to process interrupts, in a timely manner, nor to guarantee enough run-time for IO events handling.

In summary, the host fair scheduler considers the vCPU that runs both IO and CPU-bound workloads to have the same priority of a vCPU running CPU-bound workload only: they both accumulate the same amount of `vruntime` in each scheduling period. As already noted, the same applies to Xen's credit2. Hence, for a vCPU with mixed workload, low IO latency cannot be guaranteed by today's VM fair schedulers. This inspires us to develop a new IO-aware VM fair scheduler, which considers that IO-bound workloads benefit from promptly preempting non-IO tasks while also benefit from not being preempted.

Table 1: ***Latency results of* `ping` *and* `seqrd` *when running in a VM with or without additional CPU-bound workload. In the overcommitted case an additional CPU-bound VM runs on the same pCPU(s).***

Type	Background pressure	Latency(ms)
ping *(non-overcommit)*	1%	0.253
ping + CPU-bound *(non-overcommit)*	100%	0.252
ping *(overcommit)*	1%	0.286
ping + CPU-bound *(overcommit)*	100%	**2.159**
seqrd *(non-overcommit)*	2%	0.367
seqrd + CPU-bound *(non-overcommit)*	100%	0.367
seqrd *(overcommit)*	2%	0.374
seqrd + CPU-bound *(overcommit)*	100%	**7.167**

2.5 Target Workloads

Remarkably, when overcommitting multiple VMs hosting CPU-bound workloads only, their performance degrades due to the time-sharing of hardware resources – which are limited. Anubis *does not* target this use-case. Anubis is specifically designed to target VMs that run both IO and CPU-bound workloads, a very common use-case in Cloud, which includes Live streaming transcoding, Web server, Storage service, Email server, etc. [6–10, 66]. Those are characterized by an IO workload frequently requiring substantial compute.

Here we use two basic examples – Linux's `ping` command, and `sysbench-seqrd` [64], to clarify what issues we aim to solve. In the overcommitted case, a target vCPU is co-running on a pCPU together with a vCPU running exclusively a CPU-bound workload. In this context, the target vCPU experiences the added computing pressure from the other (background) vCPU. The target vCPU runs `ping` or `sysbench-seqrd` together with or without an additional CPU-bound workload.

As shown in **Table 1**, in the non-overcommit case or the overcommitted case without CPU-bound workload, the fair scheduler can schedule the vCPU properly: the latency remains low. However, in the overcommitted scenario where the vCPU runs an IO and a CPU-bound task, the performance of the IO task is *heavily affected*: 8x and 19x longer latency.

3 Analysis of Previous Works

Previous works proposed various solutions to mitigate the IO performance degradation due to vCPU inactivity in overcommitted scenarios. We organize such works into 3 categories, each of which is discussed below. **Table 2** summarizes previous works, highlighting what software modifications they require, what VMM they target, and their key limitations.

3.1 vCPU Inactivity Period Reduction

vSlicer extends Xen's *credit* scheduler [69] – a former version of *credit2*. vSlicer [70] improves the vCPU responsiveness by increasing the task context switch rate during the IO processing – shortening the scheduling timeslice, which successfully reduces the IO latencies despite VM overcommitment. The tail latency will be lower because the vCPU inactivity period is shorter. However, the vCPU will still be

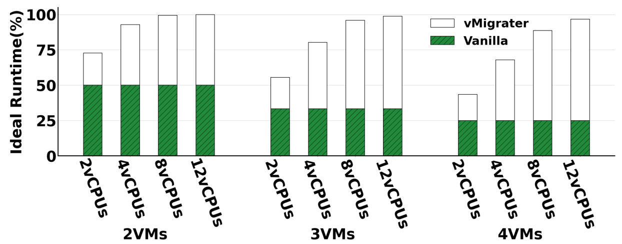


Figure 4: *vMigrater IO process maximum extendable runtime, 2/4/8/12 vCPUs VMs, overcommit ratio 2/3/4:1.*

preempted at any time, which deteriorates the worst-case latency as discussed in the **§ 2.3**. Hence, two timeslices may elapse before an interrupt is handled. Moreover, as a side effect, the shorter the vCPU inactivity period, the higher the number of expensive VM context switches – which reduces the total amount of work doable by VMs in the unit of time.

Similarly to vSlicer, AQL_sched [65] adjusts the scheduling interval of the vCPU to achieve better performance. AQL_sched categorizes each vCPU by profiling the workload running on the vCPU. AQL_sched associates a workload type with each vCPU and schedules vCPU with the best "quantum length" according to their type. However, the quantum length has to be pre-configured, and it is static. If the vCPU has changed its workload type during the runtime, AQL_sched can't dynamically alter the quantum length. Additionally, AQL_sched would perform the same as vSlicer when all vCPUs are configured with the shortest quantum length. Thus, AQL_sched is affected by at least the same problems as vSlicer.

Another approach is vBalancer [15], which tries to deliver IO device interrupts only to the running vCPU(s) by redirecting interrupts at the hypervisor level. This approach can reduce the pending time of interrupt handling. In other words, it reduces the VM's interrupt response time as other non-inactive vCPUs can assist in processing interrupts quickly. However, this approach is not aware of the rescheduling IPI delivery problem mentioned in **§ 2.3**, which may lead to no interrupt response time reduction even if interrupts are redirected.

3.2 Partial Boosting

Partial boosting [40] focuses on single-CPU VMs scenarios, based on the Xen VMM. The authors introduced gray-box knowledge to try to identify IO-intensive process(es) running on a vCPU. They claimed that an IO-intensive process usually consumes few CPU resources and preempts other tasks. Thus,

Table 2: *Summary of previous works.*

Approaches	Modifies/Target VMM	Limitation(s)
Partial-Boost[40]	host kernel/Xen	Target single-core VM, unfair in SMP
vSlicer[70]	host kernel/Xen	Introduces extra context switches
vBalancer[15]	host kernel&guest software/Xen	Only address the IO device interrupt
AQL_sched[65]	host kernel/Xen	vCPU quantum is fixed after profiling
xBalloon[63]	host kernel&guest kernel/Xen	Still preempting vCPU during IO
vMigrater[35]	guest software/KVM	Only works for large vCPU size of VM

frequent context switches could indicate an IO-intensive process. Authors monitored context switches inside the vCPU by introspecting x86's `cr3` register content, which stores a pointer to a process page table. Once IO is detected on a vCPU, the priority of such vCPU is boosted by putting it at the top of the scheduler queue. After the vCPU finishes its timeslice, it is moved to the end of the queue – to achieve fairness. We argue that this monitoring method is not accurate enough because the `cr3` register can only be accessed after a VM exit – i.e., when the VM is not running, which is too coarse grain to identify what specific process is responsible for IO.

3.3 Task-aware Boosting

A similar idea to partial boosting is to accurately detect and boost only the process that is doing the IO. xBalloon [63] is designed to accurately boost the IO process in a scenario where a vCPU is running both an IO process and a CPU-bound workload. xBalloon proposes to freeze the CPU-bound workload when the vCPU runs the IO process – a "balloon period". Hence, the vCPU temporally has only IO workload. As discussed in **§ 2.4**, this let such vCPU to preempt other vCPUs when an IO event is received. Hence, reducing the time from receiving to serving an interrupt. xBalloon allows the CPU-bound workload to run after the IO process yield. However, because the CPU-bound workload is run right after the IO process – fully consuming a scheduling timeslice, a fair VM scheduler will let another VM running for a timeslice. Thus, the IO process may wait a timeslice before receiving any interrupt. Therefore, the IO performance is not maximized.

The latest work is vMigrater [35]. vMigrater keeps migrating IO thread(s) between the active vCPUs. However, it is possible that most of the vCPUs of the VM are descheduled (or inactive) at the same time. **Fig. 4** shows the ideal runtime of a thread that can be achieved by leveraging vMigrater idea. We use the `perf` to trace the context switch of each vCPU, and analyze how long a thread's runtime can extend by migrating between running vCPUs until there is no active vCPU to migrate. With overcommit ratios of 2:1, 3:1, and 4:1, the native runtime of each vCPU accounts for 50%, 33%, and 25% of the total timeslice respectively. With vMigrater, for the VM with 12 vCPUs, we observed that the thread's ideal runtime can be up to 99.9%, 99%, and 97%. However, if the VM only has 2 vCPUs, the thread's ideal runtime can only extend up to 74%, 55%, and 27%. The performance improvement of the 2 vCPUs VM is minimized because most of the time, there is no active vCPU that can hold the thread. In summary, vMigrater works well only for VM with a lot of CPUs.

Another issue arises with multi-thread applications. In the worst scenario, if there is only one active vCPU, vMigrater could migrate several IO threads to that: the IO performance may deteriorate instead of improving. vMigrater suffers also from interrupt delivery delays: if the interrupt handler vCPU is inactive, the rescheduling IPI can not be delivered on time, moreover, the IO threads may have migrated to an active vCPU already. Finally, vMigrater's migrating service is in guest user-space, which could also suffer by overcommitment, and the user-space implementation makes vMigrater hard to implement in production because Cloud providers can hardly persuade clients to run extra services in their own VM.

4 Design

Anubis is based on the following ***design principles***:

- No modifications to the guest VM software – i.e., support legacy software stacks. Our solution should only modify the hypervisor to ease and maximize industry adoption.
- Exploit the knowledge on the current scheduling state of each vCPU.
- Maintain the priority of the vCPU executing IO, and accurately deprioritize it once IO ceases.
- Guarantee fairness between the boosted VM and other VMs.

Overarching Design. Anubis extends modern hypervisors fair schedulers in order to:

- **Improve the VM software responsiveness to IO events** by boosting the scheduling priority of a vCPU that receives an interrupt, or a rescheduling IPI; we exploit the hypervisor's scheduler knowledge to decide if to boost the priority or reroute an interrupt or IPI;
- **Accurately maximize the VM's IO performance** by boosting a vCPU only during the IO events period, by lightweight monitoring and introspecting each VM, in order to identify at any time if the workload on a vCPU is IO- or CPU- intensive. we introduce a heuristic to determine if an IO-intensive epoch is ongoing or ceased;
- **Maintain scheduling decision fairness** by migrating time quantas among vCPUs of the same VM – hence, introducing a new interpretation of scheduling fairness; we present an algorithm to conserve fairness and limit for how long vCPUs get priority boosted.

Anubis is designed around type-2 virtualization. However, we believe Anubis can be applied to type-1 virtualization as well.

4.1 Improve the VM Responsiveness

As mentioned before, in a resource overcommitted scenario, the vCPU inactivity time reduces the vCPU responsiveness. Latency-sensitive IO applications are largely affected by that because they can't promptly react to the arrival of IO events. Contrary to earlier work [70] that attempted to minimize the length of the vCPU inactivity period, Anubis modifies the VM scheduler to improve vCPU responsiveness. It does this by transparently observing the VM behavior to inform scheduling decisions.

To preserve the responsiveness of the software running in a vCPU, any incoming interrupt should be delivered to a vCPU with minimal delay. To achieve that, Anubis introduces *interrupt-boosting*. With *interrupt-boosting*, the hypervisor checks the destination vCPU of the interrupt. If the vCPU is running, the hypervisor injects the interrupt in it directly. Conversely, if the vCPU is not running, the hypervisor redirects the interrupt to a running vCPU of the same VM – the hypervisor will pick the one that is the most recently scheduled. If no vCPU of that VM is running, Anubis temporarily raises the scheduling priority of the destination vCPU, injects the interrupt, and wakes the vCPU up – preempting other vCPUs. When the application waiting for an IO event and the interrupt handling are on different vCPUs, a rescheduling IPI is generated to wake up the IO task. Anubis detects the "interrupt-related" rescheduling IPI, and if the destination vCPU is not running, it temporarily increases the scheduling priority of such vCPU, injects the IPI, and wakes the vCPU up immediately.

Anubis doesn't target Single-Root IO Virtualization(SR-IOV), which has the advantage of IO pass-through. It is unclear how to trace IO events with SR-IOV, a solution can be devised, but it may add (unwanted) overheads.

4.2 Maximize the VM IO Performance

Unlike the burstable VM[4, 16, 48], Anubis only boosts the vCPU during an IO event period. This may sound similar to previous work, in fact, [63] accomplishes this by modifying the guest software, but Anubis solves the same at the hypervisor level, without any guest software modification. Anubis's "accurate-boost" design tends to prevent the vCPU from being preempted when it is running IO processing. Thus, Anubis maximizes the IO performance by extending the run time of a vCPU. In other words, Anubis dynamically adjusts the CFS timeslice of vCPUs based on IO events. Hence, in order to boost a vCPU during IO event periods only, Anubis must also precisely identify vCPU's IO event periods boundaries – i.e., when a period starts and when it stops. Herein we introduce the *IO Points*: a mechanism to dynamically identify the likeliness that a vCPU is processing IO events.

4.2.1 *Accurate Detection of IO Events.*

We noticed that whenever a vCPU receives IO device interrupts or rescheduling IPIs, the vCPU will likely do IO processing. In addition, certain VM exits – specifically, the ones that read/write from/to IO device memory, also indicate IO activity. Therefore, a vCPU will gain 1 *IO Point* whenever it receives IRQ, IPI, or exits the VM with reason MMIO. For each scheduling tick, the fair scheduler will check if the *IO Points* increased compared to the previous tick. Succesfully obtaining an *IO Point* during a scheduling interval indicates that the vCPU is likely processing IO.

4.2.2 *Accurate Detection of IO Events Boundaries.*

A positive balance of *IO Points* only indicates that a vCPU is/has been processing IO events. To boost the vCPU more precisely, we need to identify when IO events terminate, which allow us to stop the boosting earlier to help maintaining fairness. Our solution is inspired by how vanilla CFS handles IO tasks. In the non-virtualized scenario, an IO task will voluntarily yield CPU resources if it needs to wait for IO events. Therefore, CFS assigns higher priority to the IO task, enabling it to preempt another running task as soon as it receives an IO event.

Being able to detect the boundaries of IO event periods within a vCPU will enable us to force a vCPU to yield computing resources once an IO event period ends. We observe that we have the capability to identify and monitor the currently executing task inside a vCPU by checking the VM guest memory from the hypervisor – i.e., via introspection. If the task is different from the one we detected earlier the vCPU might have ceased IO activity. We can utilize this information to pause a vCPU earlier; thus, "shuffle back" some fairness.

4.2.3 ***Accurate Boosting Preservation.*** The actual behavior of an IO task is complicated, and the boundaries of the IO and CPU-bound workloads are not straightforward. Here, we introduce 2 terms; ***IO vCPU***: the vCPU that has IO event, and Anubis should prioritize it by preventing it from being preempted. ***Non-IO vCPU***: the vCPU that doesn't have IO event, and Anubis should deprioritize it, forcing it to voluntarily yield the computing resources if in debt. In our observation, the IO vCPU might stall the IO activity for a short time, as well as a non-IO vCPU might perform IO activity for a short time. This interference can largely hurt the performance improvement of Anubis, because it might incorrectly stop boosting an IO vCPU or start boosting a non-IO vCPU.

Anubis relies on *Confidence Points* to counterbalance the impact of unstable vCPU IO activity. *Confidence Points* record the historical IO event of vCPU, thereby helping Anubis in making accurate recognition of the IO and non-IO vCPU. The vCPU will accumulate 1 *Confidence Point* when *IO Point* has increment in the previous scheduling tick. In other words, *Confidence Points* indicate the vCPU had IO events during a previous scheduling tick interval.

If an IO vCPU has an ambiguous indication, such as no increment in *IO Points* but the running task still matches the one detected during the IO event. Anubis will decrement the current *Confidence Points* according to the *degradation policy*. Different *degradation policy* of *Confidence Points* adjust the weight of the IO event that was completed long ago. If the remaining *Confidence Points* exceed a certain threshold, Anubis will still classify it as an IO vCPU, disregarding the ambiguous evidence. On the other hand, the non-IO vCPU must continuously accumulate *Confidence Points* to exceed the threshold to be classified as an IO vCPU. This implies that Anubis will only recognize a vCPU as an IO vCPU when it is capable of upholding a steady IO event period.

4.3 Overall Fairness

Despite Anubis's ability to accurately preserving the high priority of an IO vCPU, Anubis could severely deteriorate the performance of other VMs in the system due to the possible high resource demands of the IO vCPU. Mainly because of: **1.** Anubis allows IO vCPU to promptly preempt the running task whenever an interrupt is delivered. **2.** Anubis prevents IO vCPU from being preempted, which results in an extension of the IO vCPU runtime.

More computing resources need to be consumed to boost the IO vCPU, but the computing resource is limited under the overcommitted scenario. This brought us to a trade-off: *If we want to maintain the better performance of the IO vCPU, we have to accept this unfairness.* Inspired by the policy in bare-metal scheduling, where IO and computing-intensive workloads are running together without the semantic gap of the hypervisor. The CFS prioritizes the IO workload, allowing it to preempt computing-intensive workloads in a timely manner and ensuring that it is never preempted by such workloads. As a solution to this trade-off, Anubis prioritizes the IO vCPU while reducing priority for the non-IO vCPU that from the same VM, thereby striving to uphold *overall fairness*.

We introduce Anubis's *overall fairness* design: rather than maintaining the fairness between the boosted vCPU and the other vCPUs, Anubis ensures the fairness among boosted VM and the other VMs. We introduce Anubis debts to each VM, which is shared amongst all the vCPUs associated with the VM. Whenever a VM's vCPU extends its runtime or forcibly preempts background vCPU, the vCPU adds time to the debts. In other words, Anubis boosts an IO vCPU by "borrowing" the background vCPU's time. Fairness is maintained when boosted VM pays the debts back. There are two ways to maintain fairness:

Short-term fairness: Non-IO vCPU(s) pay the debts by voluntarily yielding the computing resource to the background vCPU while the IO vCPU is boosting.

Long-term fairness: All vCPUs of the VM are IO vCPUs, the debt will keep accumulating, and none of the IO vCPUs will pay the debt during the IO event until there is a non-IO vCPU.

To prevent a VM from never paying its debt, Anubis configures a debt threshold. If the accumulated debt surpasses the threshold, the boost will stop. It is worth mentioning that the IO vCPU doesn't need to pay the debt as long as it is identified as IO vCPU, this can maintain the IO performance no worse than the vanilla case. The Anubis debt maximum limitation is configurable to provide flexibility to the Cloud provider.

4.4 Malicious IO Events and Applicability

If a malicious user generates a large amount of IO events, including rescheduling IPIs, in order to occupy system resources, Anubis' configurable limits, including the maximum debt, will stop the vCPUs of the affected VMs from getting boosted – at least until the debt has been paid off.

Anubis works best if each pCPU only has 1 IO vCPU to boost. If there is more than 1 IO vCPU running, Anubis can detect the conflict because IO vCPUs are preempting each other aggressively. Hence, Anubis will rearrange the vCPUs among available pCPUs. However, if the rearrangement doesn't work, for example because all overcommitted pCPUs have one IO vCPU already, Anubis will choose one IO vCPU to boost based on configurable policies.

4.5 Summary of Anubis' Features

Responsiveness. Anubis always ensures the responsiveness of the vCPU when an interrupt arrives. Anubis identifies IO activity in a vCPU using *IO Points*. When a vCPU receives an interrupt, Anubis ensures the vCPU can forcibly preempt the running task. The unfinished timeslice of background vCPU will accumulate to the boosted VM's debt.

Boosting. Anubis prevents IO vCPU from being preempted during the IO event period. Anubis extends the runtime of IO vCPU, and the extended time will accumulate to the debt. The IO vCPU can be preempted only if it doesn't gain any *IO Points* and *Confidence Points* is lower than the threshold, or its runtime has exceeded the maximum timeslice of the scheduler.

Accurate. Anubis accurately detects the boundaries of IO events inside vCPU. Anubis forces non-IO vCPU to yield to pay the debts. It is important to mention that, if a non-IO vCPU is doing the IO event, while its *Confidence Points* have not yet exceeded the threshold, it won't be required to pay the debts. However, the non-IO vCPU will also not be recognized as IO vCPU, so it will not be able to extend its runtime.

Fairness. Anubis provides overall fairness. Anubis checks the debt of the VM at each scheduling tick. If a VM is in debt and non-IO vCPU(s), VM will pay the debts. Anubis only lets the non-IO vCPU that is not doing IO event yield computing resources to pay the debt. The debts of VM will be reduced by the time of the unfinished part of the timeslice that the yielding vCPU was supposed to run.

5 Implementation

We implemented a prototype of Anubis based on Linux kernel 5.10.90. Our implementation contributed around ~2800 LoC atop vanilla Linux and ~1800 LoC in different automation scripts. Specifically, we mainly extended the Linux CFS scheduler and the KVM subsystem – targeting Intel x86_64 machines. Therefore, Anubis schedules only Linux tasks with the `PF_VCPU` flag, which indicates the task is a vCPU. Overall, Anubis can be instructed to schedule or not-schedule a specific VM using a provided `proc` interface. We used QEMU 6.2.0 as the user-space counterpart of the in-kernel KVM hypervisor. We purposely did not modify QEMU, which can be substituted by any other similar piece of software – broadening the applicability of Anubis to different deployments.

5.1 Interrupt Redirection and Boosting

To implement interrupt-boosting we introduced interrupt redirection and IPI boosting in Linux KVM. In Linux, each interrupt is bound to a fixed CPU or set of CPUs. For example, the disk IO interrupt is usually bound to CPU0.

We implement interrupt redirection by using Intel's logical interrupt model and rerouting the interrupt to a different vCPU at function `kvm_arch_set_irq_inatomic`. However, after version 4.14 [38, 39], Linux only supports fixed interrupt routing, missing support for the logic interrupt model. Thus, currently, several Linux developers are working on supporting the logic delivery model on recent kernels [50]. At the same time, Intel [68] stated that the non-fixed interrupt is not supported anymore in the `x2-apic` model. Anyway, in our setup, the guest kernel uses the vanilla Linux kernel 4.14.0, and in the boot-up command we have added `-nox2apic` flag to disable the `x2-apic`. Hence, the interrupt can be redirected to any vCPU.

As mentioned in **§ 2.3**, it is possible that the IO process is not running on the vCPU that receives an interrupt. The vCPU will generate a rescheduling IPI and will send it to the vCPU that has the IO process. Anubis also checks the destination vCPU of the rescheduling IPI in function `kvm_apic_send_ipi`. If the vCPU is not running, CFS will force the running task to yield to this vCPU. If there are more than 2 tasks in the CFS `runqueue`, we will use the `set_next_buddy()` function to make sure the next wake-up task is the vCPU we are trying to boost. As a result, the IO process inside the boosted vCPU is able to wake up on time. Thus, the responsiveness of vCPU is guaranteed.

5.2 Accurate Boosting

Interrupt redirection and boosting are effective for short lived IO events. But what if an IO event lasts for a long time? For example, a continuous disk read. Unlike previous works, such as [40], which guess vCPU's IO activity by observing the context switch rate, Anubis uses a set of strategies to accurately identify when a vCPU is doing IO. We describe their implementation below.

IO Points. The occurrence of the delivery of IO device interrupt or rescheduling IPI indicates a potential IO event would happen in vCPU. In x86 virtualization, an MMIO fault is signaled by a vmx `ept_misconfig` fault, indicating that the vCPU is involved in the IO event. We implemented a per-vCPU variable called *IO Points*, which is a new feature we added in each vCPU `struct task_struct`. vCPU will gain 1 *IO Point* whenever an IO device interrupts, rescheduling IPI, or KVM exit reason with MMIO occurs. Anubis compares the current vCPU *IO Points* with the previous recording in each schedule tick, implemented in `check_preempt_tick()`. Any increment of the *IO Points* indicates the vCPU had a potential IO event in the previous schedule tick.

IO Task Introspection. However, previously introduced evidence doesn't reveal the actual cessation time of the IO event. Therefore, Anubis monitors the running task of a vCPU to detect the end of the IO event. Previous work [40] relies on the `cr3` register. `cr3` register stores the address of the running task's page table directory (pgd). Changing of cr3 register indicates changing of running task. However, the guest cr3 register is only observable during the KVM exit period. The host can not inspect the running task of vCPU when VM stays in the guest mode. Instead, Anubis leverages the current_task pointer, similarly to earlier works [13, 14, 22]. The gs (per-CPU register) base address add current_task address become the per-CPU variable *current, and this pointer stores the running task address. Changing of *current indicates changing of running task. Because the current_task pointer is statically compiled, the address of current_task can be found in the kernel symbol map(System.map). The Cloud providers can easily get the address of current_task since they are the VM kernel provider.

However, Cloud providers can not acquire the System.map if clients use their customized kernel. Therefore, we have implemented a heuristic approach. Because the current_task address is a fixed address in kernel space, we can keep adding the offset of a pointer length to the gs base address and reads the value until the boundary, the next vCPU's gs base address. We will have few candidates, and most of them can be eliminated in 3 steps: **1.** current_task is a per-CPU variable, the offset to the gs base remains the same for each vCPU; **2.** current_task value change over time; **3.** current_task value is kernel space address. Initial testing shown this method working also with Kernel Address Space Layout Randomization (KASLR) or Function Granular KASLR (FG-KASLR)[46]. While Anubis mainly targets Linux guest VMs, we expect Anubis working also with other traditional OSes. For example, in FreeBSD, we could start introspecting curproc instead than current_task in Linux.

Confidence Points. Guest *current address can be mapped to host userspace via the QEMU memory_slice. Implemented in handle_ept_misconfig, the host reads the *current value during KVM exit reason for MMIO and marks the value as *IO possible.* In each schedule tick the host reads the *current again and compares the value with *IO possible.* A mismatch indicates the end of the IO event. However, the mistake is unavoidable if more than one thread is related to the IO event because each time the host only reads one value out. *Confidence Points* can help us to solve this issue.

Like *IO Point*, the *Confidence Point* is a per vCPU variable, which is a new feature we have implemented in the vCPU *struct task_struct.* In each schedule tick, vCPU can get 1 *Confidence Point* if vCPU has gained *IO Points* in the previous schedule tick, implemented in check_preempt_tick(). The higher the *Confidence Points*, the more IO activities have been done previously for this vCPU. In contrast, the non-IO vCPU has to continue gaining *Confidence Points* until it is larger than the threshold to be considered as an IO vCPU, as IO vCPU is supposed to maintain a steady IO event.

The current degradation policy of Anubis in using is to divide *Confidence Points* by two. For example, with a threshold set as four, if an IO vCPU has a steady IO event in the previous 100 schedule ticks. After the IO event is completed, this IO vCPU will lose all *Confidence Points* in five ticks, and be recognized as a non-IO vCPU. On the contrary, a non-IO vCPU needs a steady IO event lasting at least four schedule ticks to be considered as an IO vCPU. Higher *Confidence Points* threshold indicates harder to recognize a vCPU as an IO vCPU.

Confidence Points degradation policy and the *Confidence Points* threshold are fully configurable.

5.3 Anubis Debt System

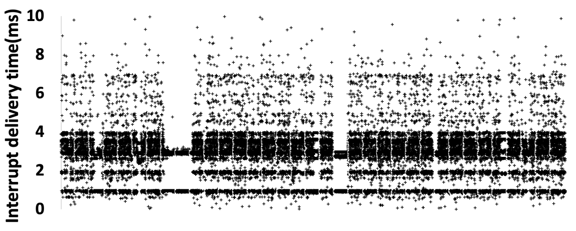
Anubis maintains fairness among VMs. We implemented a feature in each struct kvm structure called debt. Unlike the current burst VM used in the Cloud [4, 16, 48], the Anubis debt gives more granularity by boosting vCPU solely during the IO event period, rather than being constantly active. Each vCPU of the VM will contribute to the debts by accumulating the time gained from other backgrounds vCPU, and vCPUs of the same VM can synchronize access to this shared variable, controlled by spin-lock. Whenever a vCPU preempts another vCPU or extends its runtime, the vCPU will accumulate the "borrowed" time to its debts.

Anubis checks the debts at every schedule tick, implemented in check_preempt_tick(). Anubis set a *Maximum Debt* threshold to prevent the IO vCPU boosting endlessly. If the total debt reaches such threshold, the IO vCPU boost will be disabled, it can't preempt other vCPU when an interrupt is delivered or maximize the IO performance by extending its runtime when it gains *IO Points.* The*Maximum Debt* threshold is fully dynamically configurable, for example, a Cloud provider can set the threshold based on its cost structure.

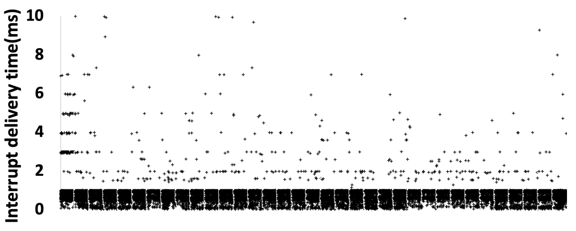
5.4 Anubis Policies

Anubis provides configurable policies for worst-case scenarios. When an IO vCPU is preempted by another IO vCPU (both *Confidence Points* are larger than the threshold), Anubis recognizes the pCPU has more than one IO vCPU. Anubis first tries to rearrange the vCPUs by switching the core affinity of a IO vCPU with another non-IO vCPU of the same VM. If the rearrangement doesn't work, for example because all vCPUs of the VM are IO vCPU, a different policy must be chosen. Currently, Anubis only boosts the IO vCPU that has the highest *Confidence Points.* However, the decision is configurable; for example, the IO vCPU that has the lowest debts will be boosted, or no vCPU is boosted.

In summary, to gain better performance Anubis can tolerate the aggressive preemption triggered by interrupt delivery, which happens between multiple IO vCPUs. However, due to limited compute resources in the overcommitted scenario, Anubis allows only one IO vCPU to maximize its IO performance by extending its runtime.



(a) Vanilla result



(b) Anubis result

Figure 5: ***Time interval between KVM delivered interrupt and receiver vCPU is active, 4 vCPUs VMs, overcommit ratio 2:1***

6 Evaluation

Testbed. We run experiments on a custom-built server based on the Supermicro X11DPi-N motherboard with a total of 768GB of DRAM and two Intel(R) Xeon(R) Gold 6230R CPUs at 2.1GHz – 26 cores each. While supported by Anubis, hyper-threading is disabled for more consistent results. For the same reason, all experiments are run with the CPU performance profile – disabled frequency scaling, and disabled turbo boost. VMs have 8GB of RAM, and between 2 and 8 vCPUs. Each guest runs Ubuntu 18.04 LTS with vanilla (unmodified) Linux kernel 4.14.0-041400-generic. The host runs Ubuntu 22.04 LTS with Linux kernel 5.10.90 extended with Anubis. The host kernel scheduling frequency is $1kHz$, so the scheduling tick is set to $1ms$, and the scheduler period (`sched_latency_ns`) is set to $8ms$ as recommended by Red Hat [28].

We considered VM overcommitted ratios of 2:1, 3:1, and 4:1 vCPUs to pCPU, which we believe are the most relevant for the state of the practice, as most of the elastic Cloud specifications [3, 11, 17, 18, 25, 26, 32, 43, 54, 67] – but Anubis put no limitations on the actual overcommitted ratio. In our experiments, all the pCPUs reside in the same NUMA node, and we divided those pCPUs into two pools [53]. A computing pool, depending on the vCPUs size of the VM we test, 2 to 8 pCPUs, which hosts all vCPUs – with a 2:1, 3:1, or 4:1 overcommitted ratio; and another pool, 2 to 8 pCPUs hosting anything else, including all vHost and QEMU-IO threads. This ensures that only vCPUs contend for the same pCPU resource.

Benchmarks. **Table 3** lists the real-world workloads we used in the evaluation. We run the same computing-intensive workload, sysbench-cpu benchmark[64], for all the VMs to create background pressure, and one VM runs the IO-intensive workload as the test case. Our upper bound is the non-overcommit case: each vCPU of the VM is running solely on a pCPU.

Table 3: ***Macro-benchmarks test set***

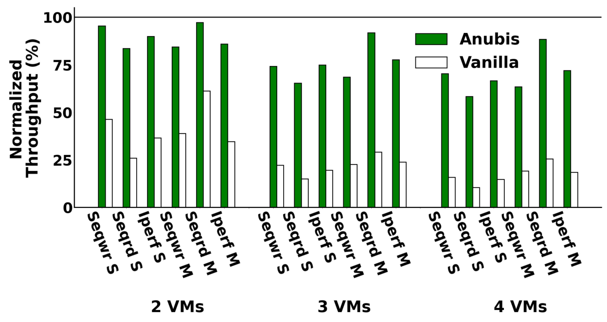
Application	Description
HBase	Sequential scan records with YCSB [72].
HDFS	Sequential read 26GB with TestDFSIO [27].
MySQL	`seqrd` and `seqwr` 10GB files with sysbench [64].
MongoDB	Insert with MongoDB-performance-test [56].
Nginx	Concurrent requests with ApacheBench [5].
Postmark	Simulate mail servers files operations [57].
Redis	Set values with Redis-benchmark [60].

Experiments. We evaluated how **§ 6.1**: Anubis shortens the vCPU inactivity time; **§ 6.2**: Anubis improves the VM's IO performance under the overcommit scenario; **§ 6.3**: Anubis maintains short-term and long-term fairness; but also **§ 6.4**: Anubis overheads; **§ 6.5**: Anubis vs latest work; and **§ 6.6**: Anubis in serverless computing scenarios.

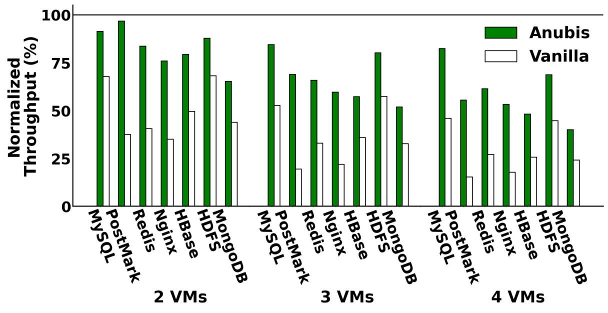
6.1 vCPU Responsiveness

Boosting based on device interrupt and rescheduling IPI, together with the interrupt redirecting improve the responsiveness of the vCPU. We use Linux kernel event tracepoint to help us understand how Anubis can help reduce interrupt pending time. For each coming interrupt(device interrupt or rescheduling IPI); When the KVM delivered the interrupts to a vCPU we recorded the first timestamp at function `__apic_accept_irq`. Then we record the time when receiver vCPU is active by tracing the scheduler event tracepoint `sched:sched_switch`. We measure the time interval between those 2 events. If the interrupt receiver vCPU is active when the interrupt is delivered, the time interval would be 0 because the vCPU at that time is active to handle the interrupt. In this experiment, we use 2 VMs, each has 4 vCPUs and overcommit at 4 pCPUs. We choose the Nginx server as the IO-intensive workload, and both VMs also run sysbench-cpu as the computing-intensive workload. In total, we have measured about ~35000 interrupts. The experimental results are presented in **Fig. 5**. The y-axis represents the time interval expressed in milliseconds, and the x-axis denotes each interrupt measurement.

The vanilla case results are depicted in **Fig. 5a**. Here, we observe that the average interval time between the delivery of the interrupt and the vCPU resuming operation is approximately 3-4 *ms*. In the worst-case scenario, expected as we analyzed in **§ 2.4**, the interrupt come right after the vCPU got preempted. This duration can extend up to about 7-8 *ms*, equivalent to roughly 1 schedule timeslice of a task entity on CFS in our setup. Additionally, if the IO thread is not executing on this vCPU, an extra rescheduling IPI is necessitated. In the worst-case scenario, a maximum of 14-16 *ms*(twice the maximum delay) must pass before the VM can respond to the interrupt. Notice that we only trace the time interval of the individual interrupt, the result in Fig doesn't show the accumulated delay time of the consecutive IO device interrupts + rescheduling IPI delivery mentioned above.



(a) *Micro. "S" single thread, "M" multi-threads)*



(b) *Macro*

Figure 6: ***Benchmarks results, normalized to the non-overcommit case, 4 vCPUs VMs, overcommitted ratio of 2/3/4:1***

On the contrary, **Fig. 5b** presents the results of Anubis, which adopts interrupt redirecting and boosting, the vCPU resumes operation in a timely manner and is capable of responding to the interrupt event within 1 *ms* of its delivery in most instances. Although this does not guarantee that the active vCPU will process the interrupt immediately after the recipient, the unavoidable delay time is trimmed by about 350% on average for each individual interrupt.

6.2 IO Performance

Here we tested Anubis's performance improvement compared with the vanilla case for both disk IO and network IO applications. Our upper bound is the non-overcommit case, in which the vCPU is solely running on a pCPU. Results show in **Fig. 6**.

Micro-benchmark. We conducted a series of disk read and write tests using sysbench-fileio[64], and iPerf3[33] for network IO testing. The results are displayed in **Fig. 6a**, and normalized to the non-overcommit case. With Anubis enabled, on average, there is a notable performance improvement when compared to the vanilla case, ranging from 59% to 500%, and the performance has improved from 45% to 97% compared to non-overcommit VM.

Macro-benchmark. We tested both single-thread and multi-thread real-world applications, including HDFS[30], LEMP[42], MongoDB[49], Postmark[57], Redis-server[59], HBase[29], Hadoop[27], and MySQL[64]. The macro application setup details and its corresponding benchmark we choose are listed in **Table 3**. Results are normalized to the non-overcommit case and present in **Fig. 6b**. On average, with Anubis, the throughputs of these benchmarks are improved by about 70%, 84%, and 103% than vanilla KVM for the settings with overcommitted ratios of 2:1, 3:1, and 4:1 respectively. In terms of the non-overcommit case, the Anubis improves the performance up to 40% to 97% of the non-overcommit VM performance.

Both micro and macro benchmarks show that with a higher overcommitted ratio, on average the degree of improvement relative to the vanilla case remains approximately constant. However, achieving the performance of non-overcommit cases becomes increasingly difficult. This can be attributed to Anubis's design, which takes care of fairness among the VMs.

Different number of vCPUs. We run VMs with different numbers of vCPUs to demonstrate that Anubis can still meet the promised performance improvements. The experiments run the same macro benchmark set, with an overcommitted ratio of 2:1, and each VM with 2, 4, and 8 vCPU, the results show in **Fig. 7**. We can observe that the different numbers of vCPUs do not affect the performance improvement of Anubis. Additionally, the performance improvement of Anubis can be even better if the VM has many vCPUs. Because the more non-IO vCPUs, the easier the fairness to achieve, Anubis can boost the IO vCPU more aggressively. On average, the performance improvement remains about 38% to 500% compared with the Vanilla case, and 60% to 97% compared with the upper bound non-overcommit case.

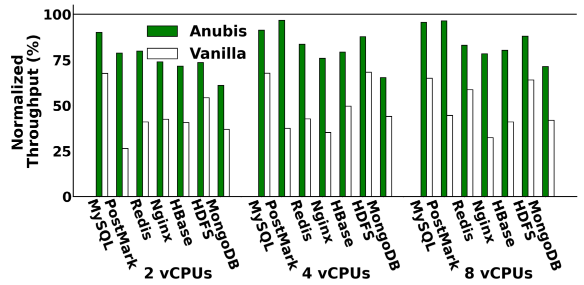


Figure 7: ***Macro-benchmark, normalized to non-overcommit case, 2/4/8 vCPUs, overcommitted ratio 2:1***

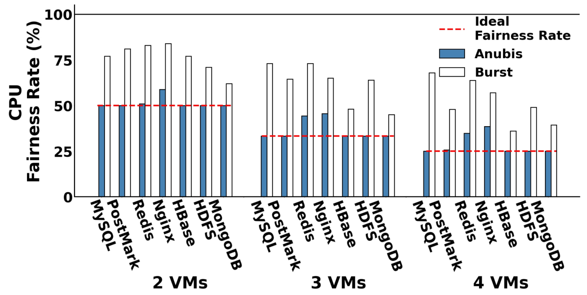
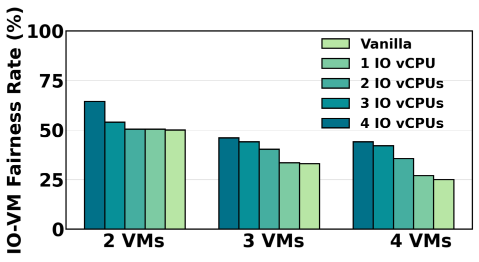
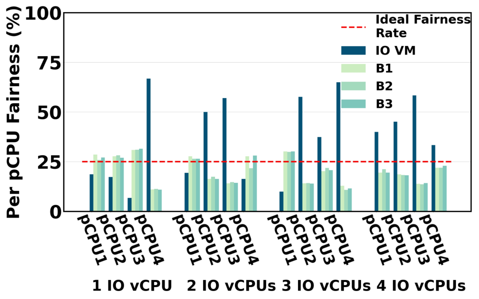


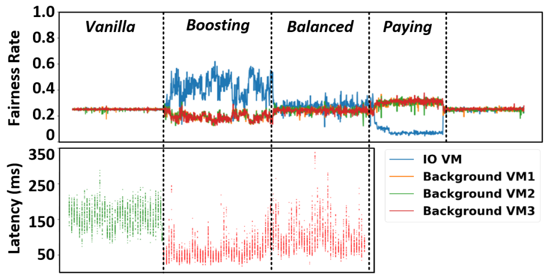
Figure 8: ***Fairness of Burst VM to achieve the Anubis performance, 4 vCPUs, overcommitted ratio 2/3/4:1***



(a) *Short-term fairness, different number of IO vCPU*



(b) *Short-term fairness, per pCPU in detail*



(c) *Long-term fairness and performance in different stages*

Figure 9: *Anubis fairness evaluation*

6.3 Overall Fairness

This section illustrates how Anubis guarantees fairness among overcommitted VMs. **§ 6.3.1** show that Anubis is not a simple boost like burst VM. **§ 6.3.2** illustrate how Anubis ensure short-term fairness **§ 6.3.3** describe how Anubis guarantee long-term fairness. In the following experiments, we use `perf` to trace the context switch for each vCPU task, and based on that, we can correctly accumulate the actual execution time of each vCPU task.

6.3.1 ***Anubis vs Burst.*** To substantiate the delta of our approach versus a naive boost solution, we set up an experiment to illustrate how Anubis is different compared to the burst VM in Cloud [4, 16, 48]. **Fig. 8** shows the fairness results of the 4 vCPU VM under-overcommit and running the same macro benchmark sets. The performance results are the same as **Fig. 6b**. The most interesting part is the white bar, the *Burst*, which indicates how many CPU resources the overcommitted vanilla VM needs to burst to achieve the same performance as Anubis. With the same performance improvement, Anubis offers a nearly ideal fairness rate, which proves that Anubis can accurately detect and boost the IO vCPU.

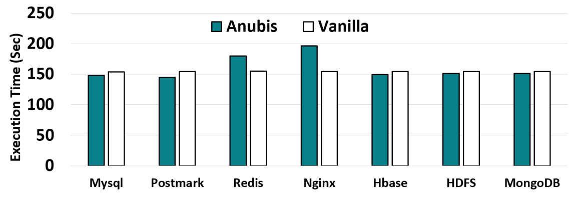
6.3.2 ***Ensure Short-term Fairness.*** As mentioned in **§ 4.3**, non-IO vCPUs can yield computing resources(paying debts) while the IO vCPU is boosting(borrowing debts). The more non-IO vCPUs, the better the fairness can be achieved. Thus, we have picked single-thread and multi-thread IO-intensive workloads to show the difference. We use a 4 vCPUs VM, and we show the fairness between the boosted VM and the background VMs with overcommitted ratio from 2:1 to 4:1.

Fig. 9a presents the overall fairness of the boosted VM during the Boosting time. The 4 cases are as follows: *1-IO-vCPU:* The sysbench-seqrd thread is pinned to the vCPU that handles the interrupt. *2-IO-vCPU:* The redis-server thread is pinned to the vCPU that doesn't handle the IO device interrupt, while the vCPU handling the interrupt is also marked as an IO vCPU. *3-IO-vCPU and 4-IO-vCPU:* each PHP worker and the Nginx server is pinned to a vCPU, while *3-IO-vCPU* case we leave 1 vCPU as non-IO vCPU.

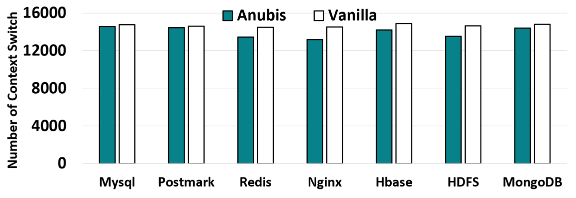
In the scenario of *1-IO-vCPU*, the fairness across VMs is closely related to the base case due to a higher number of non-IO vCPUs. However, as the number of the IO vCPU increases, short-term fairness begins to deteriorate due to the scarcity of the non-IO vCPU available to offset the debts accumulated during the Boosting stage. Detailed insight into the achievement of short-term fairness is depicted in **Fig. 9b**. Which presents CPU resource usage per vCPU in the case of an overcommitted ratio of 4:1. Observations reveal that Anubis facilitates non-IO vCPUs in yielding CPU resources to background vCPUs, while correctly boosting IO vCPUs.

Despite these insights, it's worth noting that short-term fairness can be better. In our experimental setup, we created a worst-case distributed scenario by forcibly assigning the IO threads to different vCPUs to showcase the worst-case fairness of Anubis. In the following section, we will discuss how Anubis can maintain long-term fairness, especially in the context of multi-thread IO-intensive workloads.

6.3.3 ***Guarantee Long-term Fairness.*** In the Anubis debts system, there are 3 stages during the execution time. **Boosting**: The debts are increasing, IO vCPUs are demanding more running time and keep borrowing debts, and less payback from non-IO vCPU. **Balanced**: The debts are dynamically stable, with no increase or reduction. This could happen either by debts having reached the maximum limitation or enough payback of the debts by the non-IO vCPUs. **Paying**: The debts are reducing, and non-IO vCPUs pay more debts than IO vCPUs borrow, which usually happens when IO event is finished. In this experiment, we use Nginx server as the test benchmark as it is a complicated multi-thread IO-intensive workload.



(a) ***Background computing-intensive workload execution time, measured during Boosting stage***



(b) ***Number of Context Switch, measured during Boosting stage***

Figure 10: ***Anubis overheads evaluation***

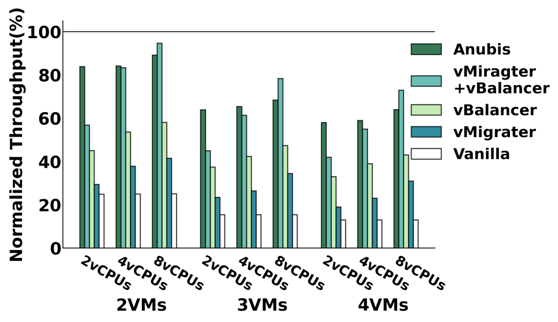


Figure 11: ***Anubis vs Previous works***

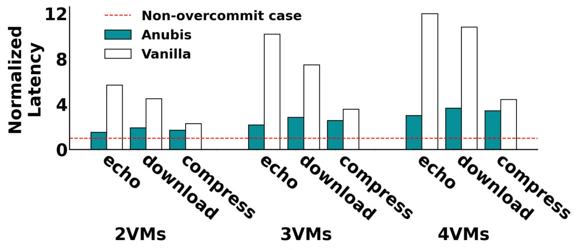


Figure 12: ***Anubis + FaaS(OpenLambda) evaluation***

Fig. 9c shows how the VM fairness and the Nginx performance change during the Boosting, Balanced, and Paying stages. The x-axis is sampled every 100 *ms*, we used `perf` to record the context switch and accumulate the actual runtime of each vCPU within the 100 *ms* interval to generate the fairness rate. Initially, we execute the Nginx server [51] under the vanilla case, recording per-request latency from the Apache benchmark [5]. In the middle of execution, Anubis is activated, resulting in a boosted state for the VM. In return, the latency is reduced during the Boosting stage. To avoid significant unfairness, the VM's boosting is curtailed once the accumulated debt hits the maximum limitation, marking the start of the Balanced stage. Owing to the IO vCPU's exemption from paying debt, the performance declines but remains **no worse** than the vanilla case. Following the completion of all requests by the Nginx server, Anubis identifies the absence of any IO event. The IO vCPUs change to non-IO vCPUs and start to pay back the debt by yielding computing resources to the background VMs. The experiment proves that Anubis can guarantee long-term fairness.

6.4 Overhead Assessment

We check the background VM workload execution time to show the affection of Anubis to the background VM. The experiments are running 4 vCPUs VM with an overcommitted ratio of 4:1, and we run the same macro benchmarks set to show the performance impact on the background VM. The y-axis denotes the execution time of the background computing-intensive workload(a multi-thread prime number calculator). All the experiments' measurements happen in the Boosting stage because we believe if we measure in the long-term, the execution time would not be affected as shown in **Fig. 9c**. Results shown in **Fig. 10a**, in most cases, the background VM doesn't get affected. However, if the boosted IO workload has many IO vCPUs (Nginx case), the performance of the background VM would drop to about 75% compared with the vanilla case.

In order to justify that Anubis does not bring overheads by increasing the expensive VM context switch like the previous work [70], we use the `perf` tool to accumulate the total context switches of each VM and compare it with the vanilla case under different overcommitted ratios. Shown in **Fig. 10b**, the number of context switches is reduced in several cases. After analysis, we believe it is because the IO vCPU can escape from being preempted whenever it can gain *IO Points* until the maximum timeslice. The more IO vCPUs(Nginx case), the fewer the context switch would happen as the IO vCPU would keep running. For other cases, the number of context switches remains the same compared to the vanilla case.

We further use `perf` tool to measure the cost of Anubis introspection during each scheduling interval (every 1 *ms*). The cost is about 150 cycles, which is trivial, and we acknowledge that there is still space for further optimization.

6.5 Anubis vs Previous Works

We compared Anubis with vanilla Linux, vBalancer [15], vMigrator [35], and the combination of vMigrator with vBalancer – this is because vMigrator must run with vBalancer to enable interrupt redirection. We primarily compared with the result of vMigrator because, in their paper, they have compared with other approaches, vSlicer[70], xBalloon[63], and none of them have better performance compared with the vMigrater. We didn't use the open-source code from the vMigrater because it does need kernel modifications from vBalancer, unlike what we originally understood from the paper [1]. Because the interrupt redirection is part of the Anubis design, we can emulate the vBalancer by just enabling the IO device interrupt redirection of Anubis.

In this experiment, we ran VMs that have different vCPU sizes from 2 to 8 vCPUs, and with the overcommitted ratio from 2:1 to 4:1, the performance result is shown in **Fig. 11**.

As we explained previously, if we run the vMigrater only, the performance of the IO application will **not** be increased because the vCPU that receives the IO device interrupt will not send the rescheduling IPI if it is not running. Additionally, as **Fig. 4** shows, under the overcommitted ratio of 2/3/4:1, with the 2 vCPU VM, the IO thread runtime can only extend up to 74%/54%/47% of maximum timeslice. While in the 4 vCPUs and 8 vCPUs VM, the IO thread ideal extendable runtime can extend up to 90%/78%/60% and 100%/95%/88% of maximum timeslice. As we expected, the experiment result shows that the improvement of vMigrater+vBalancer to the IO application is minimal in 2 vCPUs VM, and the vMigrater+vBalancer approach starts to work in the 4 and 8 vCPUs VM cases. However, Anubis offers similar results to the vMigrater approach and apply to the VMs with different number of vCPUs. While we tested our emulated vMigrater across all benchmarks, due to the lack of original kernel support, our results diverged from those reported in their paper – they are worse, making it difficult for us to validate the correctness of our results. Therefore, we only report the sysbench-seqrd as it was the most stable result we achieved. After comparing the results in their paper, which are also normalized to the non-overcommit case, we found that in some benchmarks vMigrater can achieve slightly better results than Anubis. However, vMigrater requires a VM to have many vCPUs (12 vCPUs) to accomplish this, whereas Anubis has no such prerequisites. Moreover, since Anubis doesn't necessitate modifications to the guest VM, it is more adaptable and flexible for implementation in a real Cloud environment.

6.6 Serverless Computing Scenario

Function as a Service(FaaS) is an emerging paradigm where users upload and execute small pieces of code, or *functions* in the Cloud. Major Cloud providers are already offering FaaS solutions [2, 24, 47]. A recent study of Azure Functions [61] shows that serverless functions execution time is below 10 sec for 90% functions. Given these characteristics of serverless computing, Cloud providers desire to aggressively consolidate plenty of function instances into one physical host [1].

We use the open-source OpenLambda [31] FaaS framework for our serverless computing experiment. We deployed 1 OpenLambda server per VM, configured with maximal spawns of 100 functions each time. We have registered 3 functions: echo, download, and compress services. **Echo**: The client posts data to the server, and the server returns the same data back. **Download**: We deploy a VM on another host as a storage machine connected to the OpenLambda VM. The client posts an HTTP request with an encoded descriptor to Lambda. The Lambda function will decode the descriptor and locate the target data in the storage machine. Then the Lambda function copies the target data from the storage machine to the client through the network. **Compression**: In this experiment, we are emulating a video stream service. The client posts a request to the Lambda server with the encoded video ID and level of video quality. The Lambda function first locates the video data by decoding the video ID and copying 100 picture frames of the video from the storage machine. Then the Lambda function will compress all frames with the requested quality and send them to another storage machine.

[1]We did exchange several emails with the authors to reach such a conclusion. Moreover, we were unable to get a working code from the authors.

We have measured the latency of each request and compared the result to both vanilla and non-overcommit cases, and the results are in **Fig. 12**. We can see that the compression service gets lower improvement compared with the Echo service. Because the compression service also includes the computing-intensive workload, but Anubis only boosts the IO-intensive part of the service.

7 Conclusion

This paper identifies why low-latency IO cannot be guaranteed today in virtualized setups with overcommitted resources – focusing on CPU, which hasn't been adequately studied before. Previous works focus mostly on the *direct reason* causing slow IO: the vCPU containing the IO task has been descheduled. This paper spotlight the *indirect reason*: IO device interrupts or rescheduling IPIs are not immediately dispatched to vCPUs. We introduce Anubis, a new IO-aware VM scheduler that can precisely identify IO processing happening in a vCPU and accurately boost the vCPU priority during the IO processing only. Notably, Anubis does not require any guest software modification.

In summary: **1.** Anubis mitigates the impact of vCPU inactivity: instead of shortening the inactive period of a vCPU, Anubis improves the responsiveness of the vCPU by waking it up on time when there is a pending interrupt for it; **2.** Anubis maximizes the IO performance of a IO vCPU during an IO event period: instead of ensuring the priority of IO tasks inside the vCPU, Anubis maintains the IO vCPU priority by maximizing its runtime within the IO event period; **3.** Anubis ensures overall fairness among VMs while improving the IO performance: instead of maintaining the fairness of IO vCPU and other vCPUs, Anubis ensures fairness among VMs through a debt system.

The paper provides several experimental results to support Anubis' efficiency and its benefits vs state of the art. The code is available at *https://github.com/systems-nuts/Anubis.*

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation (Santa Clara, CA, USA) (NSDI'20). USENIX Association, USA, 419–434.

[2] Amazon. 2020. AWS Lambda Website. https://aws.amazon.com/lambda.

[3] Amazon. 2022. How Amazon ECS manages CPU and memory resources. https://aws.amazon.com/blogs/containers/how-amazon-ecs-manages-cpu-and-memory-resources/.

[4] Amazon. 2023. Burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[5] Apache. 2023. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[6] AWS. 2023. BBC delivers live, UHD coverage of UEFA Euros and Wimbledon with AWS. https://aws.amazon.com/cn/blogs/media/bbc-delivers-live-uhd-coverage-of-uefa-euros-and-wimbledon-with-aws/.

[7] AWS. 2023. Explore.org live streams nature cams to global audiences with AWS. https://aws.amazon.com/cn/blogs/media/explore-org-live-streams-nature-cams-to-global-audiences-with-aws/.

[8] AWS. 2023. LAMP Server on AWS. https://aws.amazon.com/marketplace/pp/prodview-gqnnpbafrkkys.

[9] AWS. 2023. Partner Success with AWS. https://aws.amazon.com/partners/success/.

[10] AWS. 2023. Washington Post's Arc publishing platform uses AWS to transform the broadcast landscape. https://aws.amazon.com/cn/blogs/media/washington-posts-arc-publishing-platform-uses-aws-to-transform-the-broadcast-landscape/.

[11] Blueprint. 2022. https://blueprints.launchpad.net/nova/+spec/nova-change-default-overcommit-values.

[12] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. 2018. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (Boston, MA, USA) (USENIX ATC '18). 85–96.

[13] Kevin Burns, Antonio Barbalace, Vincent Legout, and Binoy Ravindran. 2014. KairosVM: Deterministic introspection for real-time virtual machine hierarchical scheduling. In Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA). 1–8. https://doi.org/10.1109/ETFA.2014.7005061

[14] Kevin Burns, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. 2019. PrVM: A Multicore Real-Time Virtualization Scheduling Framework with Probabilistic Timing Guarantees. SIGBED Rev. 16, 3 (nov 2019), 14–20. https://doi.org/10.1145/3373400.3373402

[15] Luwei Cheng and Cho-Li Wang. 2012. VBalance: Using Interrupt Load Balance to Improve I/O Performance for SMP Virtual Machines (SoCC '12). Association for Computing Machinery, New York, NY, USA, Article 2, 14 pages. https://doi.org/10.1145/2391229.2391231

[16] Huawei Cloud. 2023. Elastic Cloud Server (ECS). https://www.huaweicloud.com/intl/en-us/product/ecs.html.

[17] Huawei Cloud. 2023. A Summary List of x86 ECS Specifications. https://support.huaweicloud.com/intl/en-us/productdesc-ecs/ecs_01_0014.html.

[18] Key concepts and definitions for burstable performance instances. 2023. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html.

[19] Mehiar Dabbagh, Bechir Hamdaoui, Mohsen Guizani, and Ammar Rayes. 2015. Efficient datacenter resource utilization through cloud resource overcommitment. In 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). 330–335. https://doi.org/10.1109/INFCOMW.2015.7179406

[20] Xiaoning Ding, Phillip B. Gibbons, and Michael A. Kozuch. 2013. A Hidden Cost of Virtualization When Scaling Multicore Applications. In 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13). USENIX Association, San Jose, CA. https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/ding

[21] 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). USENIX Association, Philadelphia, PA, 73–84. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ding

[22] Michael Drescher, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. 2016. A Flattened Hierarchical Scheduler for Real-Time Virtualization. In Proceedings of the 13th International Conference on Embedded Software (Pittsburgh, Pennsylvania) (EMSOFT '16). Association for Computing Machinery, Article 12, 10 pages. https://doi.org/10.1145/2968478.2968501

[23] Sahan Gamage, Cong Xu, Ramana Rao Kompella, and Dongyan Xu. 2014. VPipe: Piped I/O Offloading for Efficient Data Movement in Virtualized Clouds (SOCC '14). Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2671006

[24] Google. 2020. Google Cloud Functions. https://cloud.google.com/functions.

[25] Google. 2022. Get more from every core: Announcing CPU overcommit for Compute Engine. https://cloud.google.com/blog/products/compute/cpu-overcommit-for-sole-tenant-nodes-now-ga.

[26] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 845–861. https://www.usenix.org/conference/osdi20/presentation/hadary

[27] Hadoop. 2023. https://hadoop.apache.org/.

[28] Red Hat. 2017. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/6.0_technical_notes/deployment.

[29] HBase. 2023. https://hbase.apache.org/.

[30] Hadoop Distributed File System (HDFS™). 2023. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesi.

[31] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with openlambda. In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16).

[32] IBM. 2022. https://www.ibm.com/docs/en/cic/1.1.3?topic=SSLL2F_1.1.3/com.ibm.cloudin.doc/admintasks/configuring/customizing/allocation_ratio_templates.htm.

[33] iperf3. 2023. https://github.com/esnet/iperf.

[34] Kenta Ishiguro, Naoki Yasuno, Pierre-Louis Aublin, and Kenji Kono. 2021. Mitigating Excessive VCPU Spinning in VM-Agnostic KVM. In Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (Virtual, USA) (VEE 2021). Association for Computing Machinery, New York, NY, USA, 139–152. https://doi.org/10.1145/3453933.3454020

[35] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis C. M. Lau, Yuexuan Wang, and Yuangang Wang. 2018. Effectively Mitigating I/O Inactivity in VCPU Scheduling. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (Boston, MA, USA) (USENIX ATC '18). USENIX Association, USA, 267–279.

[36] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. 2010. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. 1–11. https://doi.org/10.1109/SC.2010.57
[37] J. Kay and P. Lauder. 1988. A Fair Share Scheduler. Commun. ACM 31, 1 (jan 1988), 44–55. https://doi.org/10.1145/35043.35047
[38] Linux kernel. 2023. https://elixir.bootlin.com/linux/v4.14.325/source/arch/x86/kernel/apic/apic.c.
[39] Linux kernel. 2023. https://elixir.bootlin.com/linux/v4.14.325/source/arch/x86/kernel/apic/apic_flat_64.c.
[40] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. 2009. Task-Aware Virtual Machine Scheduling for I/O Performance. (VEE '09). Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/1508293.1508308
[41] Linux KVM. 2023. https://www.linux-kvm.org/page/Main_Page.
[42] LEMP. 2023. https://lemp.io/.
[43] Scott D. Lowe. [n. d.]. Best Practices for Oversubscription of CPU, Memory and Storage in vSphere Virtual Environments. Dell.
[44] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. 2015. VFair: Latency-Aware Fair Storage Scheduling via per-IO Cost-Based Differentiation (SoCC '15). Association for Computing Machinery, New York, NY, USA, 125–138. https://doi.org/10.1145/2806777.2806943
[45] Hui Lu, Cong Xu, Cheng Cheng, Ramana Kompella, and Dongyan Xu. 2015. vHaul: Towards Optimal Scheduling of Live Multi-VM Migration for Multi-tier Applications. In 2015 IEEE 8th International Conference on Cloud Computing. 453–460. https://doi.org/10.1109/CLOUD.2015.67
[46] LWN. 2011. https://lwn.net/Articles/444503/.
[47] Microsoft. 2020. Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions.
[48] Microsoft. 2023. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable-workload-example.
[49] MongoDB. 2023. https://www.mongodb.com.
[50] Ricardo Neri. 2022. https://www.spinics.net/lists/kernel/msg4348466.html.
[51] Nginx. 2023. https://nginx.org/.
[52] Diego Ongaro, Alan L. Cox, and Scott Rixner. 2008. Scheduling I/O in Virtual Machine Monitors. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (Seattle, WA, USA) (VEE '08). Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1346256.1346258
[53] OpenEuler. 2022. https://docs.openeuler.org/en/docs/20.03_LTS_SP2/docs/Virtualization/best-practices.html.
[54] Openstack. 2022. https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html.
[55] Oracle. 2023. https://www.oracle.com/uk/a/ocom/docs/why-kvm-is-winning.pdf.
[56] Mongodb performance test. 2023. https://github.com/idealo/mongodb-performance-test.
[57] PostMark. 2023. https://www.filesystems.org/docs/auto-pilot/Postmark.html.
[58] Xen Project. 2023. https://xenproject.org/.
[59] Redis. 2023. https://redis.io/.
[60] Redis-benchmark. 2023. https://redis.io/docs/management/optimization/benchmarks/.
[61] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20). USENIX Association, USA, Article 14, 14 pages.
[62] Jianchen Shan, Weiwei Jia, and Xiaoning Ding. 2017. Rethinking Multicore Application Scalability on Big Virtual Machines. In 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS). 694–701. https://doi.org/10.1109/ICPADS.2017.00094
[63] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis C. M. Lau. 2017. Preserving I/O Prioritization in Virtualized OSes. In Proceedings of the 2017 Symposium on Cloud Computing (Santa Clara, California) (SoCC '17). Association for Computing Machinery, New York, NY, USA, 269–281. https://doi.org/10.1145/3127479.3127484
[64] sysbench. 2023. https://github.com/akopytov/sysbench.
[65] Boris Teabe, Alain Tchana, and Daniel Hagimont. 2016. Application-Specific Quantum for Multi-Core Platform Scheduler. In Proceedings of the Eleventh European Conference on Computer Systems (London, United Kingdom) (EuroSys '16). Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/2901318.2901340
[66] Twitch. 2023. https://blog.twitch.tv/en/2017/10/10/live-video-transmuxing-transcoding-f-fmpeg-vs-twitch-transcoder-part-i-489c1c125f28/.
[67] Vmware. 2022. https://www.reddit.com/r/vmware/comments/dl2bt8/do_you_overcommit_cpu_in_your_environement/.
[68] Intel® 64 Architecture x2APIC Specification. 2023. https://www.naic.edu/~phil/software/intel/318148.pdf.
[69] xen. 2013. https://wiki.xenproject.org/wiki/Credit2_Scheduler.
[70] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. 2012. VSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing (HPDC '12). Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2287076.2287080
[71] Cong Xu, Brendan Saltaformaggio, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. 2015. VRead: Efficient Data Access for Hadoop in Virtualized Clouds. In Proceedings of the 16th Annual Middleware Conference (Vancouver, BC, Canada) (Middleware '15). Association for Computing Machinery, New York, NY, USA, 125–136. https://doi.org/10.1145/2814576.2814735
[72] Yahoo. 2023. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.
[73] Olive Zhao. 2021. https://forum.huawei.com/enterprise/en/why-are-huawei-cloud-computing-products-switched-from-xen-to-kvm/thread/818617-893.