Preserving I/O Prioritization in Virtualized OSes

Kun Suo[1], Yong Zhao[1], Jia Rao[1], Luwei Cheng[2], Xiaobo Zhou[3] and Francis C. M. Lau[4]
[1]*University of Texas at Arlington, {kun.suo, yong.zhao, jia.rao}@uta.edu*
[2]*Facebook, chengluwei@fb.com*
[3] *University of Colorado, Colorado Springs, xzhou@uccs.edu*
[4]*The University of Hong Kong, fcmlau@cs.hku.hk*

ABSTRACT

While virtualization helps to enable multi-tenancy in data centers, it introduces new challenges to the resource management in traditional OSes. We find that one important design in an OS, prioritizing interactive and I/O-bound workloads, can become ineffective in a virtualized OS. Resource multiplexing between multiple tenants breaks the assumption of continuous CPU availability in physical systems and causes two types of priority inversions in virtualized OSes. In this paper, we present xBalloon , a lightweight approach to preserving I/O prioritization. It uses a balloon process in the virtualized OS to avoid priority inversion in both short-term and long-term scheduling. Experiments in a local Xen environment and Amazon EC2 show that xBalloon improves I/O performance in a recent Linux kernel by as much as 136% on network throughput, 95% on disk throughput, and 125x on network tail latency.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Operating systems**;

KEYWORDS

Virtualization, Cloud Computing, Multi-tenancy, Semantic Gaps.

1 INTRODUCTION

Due to its support for multi-tenancy, virtualization is becoming ubiquitous in datacenters. Popek and Goldberg's virtualization requirements [37] suggest that a program running in virtualized environments should exhibit a behavior essentially identical to that in physical environments. This property does not hold for many programs, especially those equipped with their own resource management, e.g., operating systems (OSes). The culprit is the *semantic gap* between physical and virtual environments. Virtualization presents the illusion of dedicated hardware, but resources are often multiplexed among users and have discontinuous availability.

The semantic gap can cause performance problems if resource management designed for physical systems becomes ineffective in virtualized OSes (a.k.a., guest OSes). I/O prioritization is an important OS design to improve system responsiveness without compromising throughput. It guarantees timely processing of important I/O events, such as interrupts, kernel threads, and user-level I/O processes, while allowing compute-bound programs to run when no imminent I/O processing needs to be handled. Thus, co-locating I/O- and compute-bound programs has been a common practice to improve system utilization. For example, interrupt handling in an OS kernel, such as packet processing, has a more paramount priority than user-level activities. Network latency is not affected by user-level computation even in a fully-loaded system. Due to I/O prioritization, interactive services can be co-located with batch jobs without suffering from long latency.

Unfortunately, the co-location of I/O- and compute-bound workloads in a virtual machine (VM) can cause severe degradation of I/O performance. In virtualized environments, CPU multiplexing/sharing and capping are widely adopted to improve resource utilization and performance isolation. For example, VMware suggests that a physical CPU (pCPU) can be shared by as many as 8 to 10 VMs [41]; AWS instances with capped CPU capacity, e.g., `m1.small`, `m1.medium`, `t1.micro`, and `m3.medium`, account for around 40% of Amazon EC2 usage at RightScale [3]; the new generation AWS T2 burstable performance instances use CPU cap to provide the baseline CPU performance [1]. Either CPU capping or sharing makes the CPU availability discontinuous to a VM. Existing studies have found that discontinuous CPU availability can delay I/O scheduling and affect TCP congestion control [19, 22, 42], interrupt handling [17] and latency-sensitive workloads [44–46] even the CPU allocation to a VM is well above the I/O demand. However, a key question remains unanswered – *why I/O prioritization in the guest OS is not doing its job*? In this paper, we discover two types of priority inversions in the guest OS when a VM runs a mixture of I/O and compute workloads and executes on discontinuous CPU.

I/O prioritization relies on two mechanisms: (1) the identification of I/O-bound tasks [1] and (2) the preemption of compute-bound tasks. Both mechanisms can be ineffective in a guest OS with discontinuous CPU. First, CPU accounting in guest OSes can be inaccurate under discontinuous time, leading to false identification of I/O-bound task as compute-bound. Second and most importantly, work-conserving (WC) scheduling, which is designed for continuous CPU availability, fails to guarantee the preemption of compute-bound tasks in discontinuous CPU. As the resources are highly consolidated and shared among VMs in virtualized environment, WC scheduling allows low priority compute-bound tasks to run when I/O-bound tasks are idle, which consumes the CPU

[1]Only user-level I/O-bound tasks need to be identified while kernel threads and interrupts inherently have a higher priority than any userspace tasks.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3127484

that would otherwise be utilized by high priority I/O tasks in the future. We call these violations of I/O prioritization *short-term* and *long-term* priority inversion, respectively.

This paper aims to preserve the priorities enforced by guest OSes in spite of CPU discontinuity. To this end, we develop xBalloon, a lightweight approach to addressing the two priority inversions. The idea is to use a CPU balloon process in the guest OS to account for the period during which CPU is unavailable to a VM. When scheduled to run, the balloon puts a VM into sleep. The balloon serves two purposes: (1) it makes CPU discontinuity visible to the guest OS to ensure correct CPU accounting; (2) by scheduling the balloon, the guest OS can autonomously decide if the VM should be paused to reserve CPU for future use.

The heart of xBalloon design is a semi-work-conserving (SWC) scheduling mode for the guest OS. To preserve I/O prioritization under constrained CPU allocation, the guest OS stays in non-work-conserving (NWC) mode by enabling the balloon process to throttle the execution of compute tasks. This is to ensure that the guest OS has sufficient resources to serve high priority I/O tasks. If the CPU allocation has slackness for low priority tasks, guest OS suspends the balloon and switches back to work-conserving (WC) mode.

We implemented xBalloon in Linux 3.18.21 and Xen 4.5.0. xBalloon is a flexible and lightweight approach that can be launched and killed as a regular Linux process. We show its effectiveness on preserving *static* and *dynamic* priorities in Linux guests. Experimental results show that xBalloon boosts the performance of various I/O workloads co-located with compute tasks and precisely preserves their priorities.

2 BACKGROUND AND MOTIVATION

In this section, we first review the assumptions in this paper, describe I/O prioritizations in Linux and discuss the causes of time discontinuity in virtualized environments. Then, we show that time discontinuity inflicts priority inversions in virtualized OSes under popular hypervisors and an OS container, which leads to degraded and unpredictable I/O performance.

2.1 Assumptions

We make the following assumptions about cloud users and typical use cases: 1) an average user would expect a virtualized OS to be fully functional and similar to a traditional OS; 2) to reduce monetary cost, users consolidate multiple, often heterogeneous workloads onto the same VM if a single workload cannot fully utilize VM resources; 3) users are unaware of the underlying resource virtualization and multiplexing, and expect task administration in the guest OS, e.g., task priorities, to be effective.

2.2 I/O Prioritization in Linux

Linux prioritizes I/O-bound tasks over compute-bound tasks in two ways. First, in-kernel I/O processing, such as interrupts and kernel threads, has inherently higher priority than user-level compute tasks. Users can also explicitly assign an elevated priority, e.g., a real-time priority, to a user-level I/O task. As such, the guest OS enforces *static* priorities between I/O and compute tasks. Second, Linux implicitly prioritizes I/O-bound tasks by enforcing *dynamic* priorities between the two. The completely fair scheduler (CFS) in Linux uses *virtual runtime* (vruntime), which tracks how much time a process has spent running on CPU, to schedule processes. CFS maintains a red-black (RB) tree-based runqueue, which sorts processes based on their vruntimes, and always schedules the process with the least vruntime. This design not only prioritizes tasks that have small vruntimes but also enforces fair CPU allocations. If an I/O task demands more than the fair share, its vruntime will not be smaller than that of the compute task and CFS assigns equal priorities to them.

2.3 Time Discontinuity

In general, there are two ways to control the CPU allocation to a VM: CPU *sharing* and *capping*. In CPU sharing, the hypervisor consolidates multiple virtual CPUs (vCPUs) on the same pCPU. VMs take turns to run on pCPUs in a weighted round robin manner. CPU capping [5] sets an upper limit on the CPU time a VM receives, e.g., T2 Instances [1] on Amazon EC2 [2]. VMs are temporarily suspended if their CPU usage exceeds the cap. The period during which a VM is not running, either due to sharing CPU with other VMs or capping, creates gaps on VM perceived time. These gaps cannot simply be overlooked by VMs. The guest OS needs to keep up with the host wall-clock time to avoid time drift. Most OSes today para-virtualize its clock to synchronize with the hypervisor. Thus, the guest OS sees discontinuous passage of time when the VM is de-scheduled and later scheduled.

2.4 Degraded I/O Performance due to CPU Discontinuity

In this section, we show that Linux fails to preserve I/O prioritization when running as a virtualized OS under discontinuous time. We emulated a scenario in which a VM is loaded with heterogeneous workloads. `Netperf` [10] and `sockperf` [14] benchmarks were used to test network throughput and latency with multiple hypervisors and an OS container. The workloads were run in one-vCPU VMs in KVM [7], Xen [18] and a Docker container [6], respectively. The VMs and container ran the Linux 3.18.21 kernel and were configured to run in two modes: *full* and *partial* CPU capacity. Under full capacity, one physical CPU (pCPU) was dedicated to the VMs or the container. Thus, CPU was always available and continuous to the guest OS. Under partial capacity, we limited the VMs or container to use 50% of the pCPU. As such, the guest OS ran for a certain period and was suspended for the same amount of time to satisfy the constraint. With partial capacity, the Linux guest experienced discontinuous CPU availability.

A `while(1)` CPU hog was used to emulate a background compute-bound task that co-ran with the I/O task in the guest OS (denoted as *I/O+CPU*). The reference I/O performance was obtained when the CPU hog was turned off (denoted as *I/O only*). Since I/O-bound tasks consume less CPU, Linux gives them higher priority than compute-bound tasks under CFS. Figure 1 (a) and (c) show that Linux effectively preserves I/O prioritization under continuous CPU. The CPU hog did not degrade network throughput and even improved tail latency. The co-location of I/O- and compute-bound tasks prevented guest OS from entering low-power states (i.e., C-states), reducing the startup cost to service network requests.

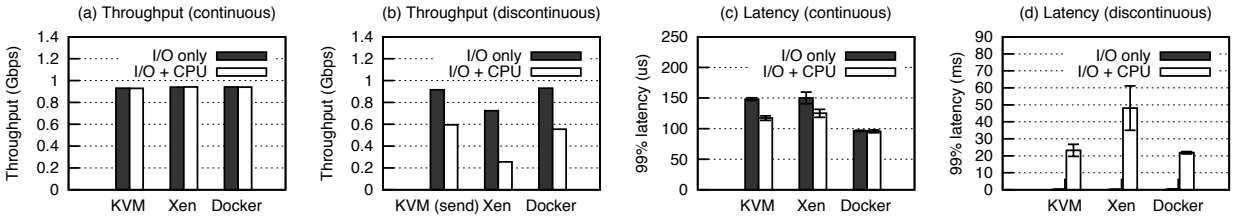

Figure 1: Network I/O suffers poor and unpredictable performance in virtualized environments. (a) and (c) Virtualization alone does not necessarily degrade I/O throughput or latency under continuous time. (b) and (d) Discontinuous time leads to significant performance loss and increasing unpredictability in VMs and containers with mixed I/O and computation.

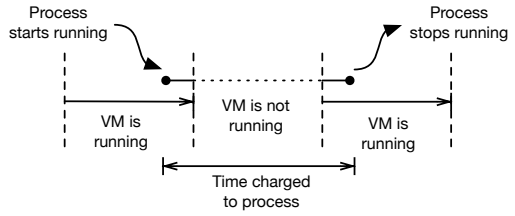

Figure 2: Time discontinuity may cause inaccurate CPU accounting to processes in the guest OS.

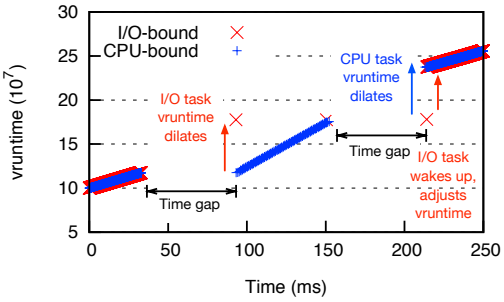

Figure 3: Time dilation only affects the scheduling of I/O-bound tasks in Linux CFS.

In contrast, both hypervisors and the container suffered significant throughput loss and latency hike under discontinuous CPU. While the reference I/O performance dropped due to reduced CPU capacity, the compute task caused up to 65% further throughput loss (i.e., Xen) and 100x latency increase with 27% variation (i.e., Xen) in 10 runs (as shown in Figure 1(b) and (d)). Network latency in Figure 1(d) in the I/O only case was in micro-seconds, thereby not showing up in the figure. These experiments clearly show that Linux guest's property of prioritizing I/O operations over compute activities is not preserved when the VMs and container experience discontinuous CPU availability.

3 ANALYZING PRIORITY INVERSIONS

3.1 Short-term Priority Inversion

Linux's CFS uses vruntime to track task CPU usage and prioritizes those with small vruntimes. Short-term priority inversion happens when the vruntimes of I/O tasks are mistakenly dilated under discontinuous time so that CFS fails to prioritize them. Figure 2 illustrates how vruntime dilation can happen under discontinuous time. If the VM is suspended right after a process starts running on a vCPU, the period in which the VM is not running will be charged to the process because the VM synchronizes its clock with the host after resuming execution. Thus, vruntime update of the process will include the time gap. Inaccurate time accounting does not affect tasks with static priority as task scheduling is not based on CPU usage. In contrast, time dilation can interfere with vruntime-based scheduling in Linux CFS.

Figure 3 illustrates how time dilation affects the scheduling of an I/O-bound task but not a compute-bound task. We co-located a `sockperf` server process with a CPU hog in a one-vCPU VM that is capped at 50% capacity of a pCPU. Figure 3 plots the vruntimes of the two tasks whenever they were scheduled or descheduled by CFS. The number of dots reflects how long a task runs on CPU. After the first time gap, the vruntime of the I/O task was dilated and it became much larger than the vruntime of the compute task. As a result, the I/O task was not scheduled by CFS until the compute task's vruntime caught up with that of the I/O task. In comparison, after the second time gap, even though the compute-bound task suffered time dilation, it was still scheduled by CFS in a fair manner. This is because CFS clamps a waking I/O task's vruntime to the minimum vruntime in the runqueue minus two time slices [31]. Therefore, the I/O task's adjusted vruntime also included the time gap. As a consequence, inaccurate time accounting only penalizes I/O tasks in CFS scheduling.

3.2 Long-term Priority Inversion

Compared to short-term priority inversion, which can be addressed by accurate CPU accounting, long-term priority inversion raises a fundamental question: *should the resource management designed for physical environments be adapted in a virtualized environment to efficiently utilize discontinuous resources?* We show that under discontinuous time work-conserving scheduling in a guest OS can lead to long-term priority inversions for I/O-bound tasks under both static and dynamic priority.

Figure 4(a)-(e) show how long-term priority inversions develop between a high priority task (e.g., I/O-bound task, white bar) and a low priority task (e.g., compute-bound task, grey bar) under discontinuous time due to a 50% CPU cap. The higher priority of the I/O task can be either statically assigned by the administrator or dynamically determined by CPU usage. Figure 4(a) shows the CPU

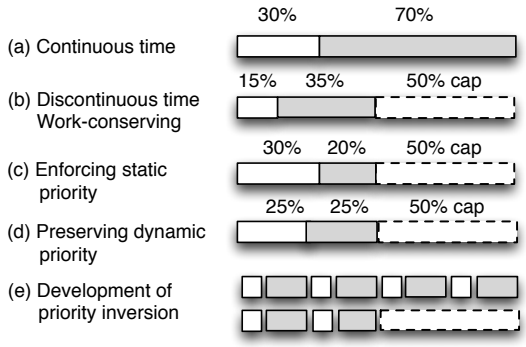

Figure 4: Work-conserving scheduling in guest OS does not preserve priorities between a high priority (white bar) and a low priority task (grey bar) under discontinuous time.

allocations between the two tasks under continuous time. The I/O-bound task consumes less CPU than the compute task thus can always preempt the latter when waking up. Note that Figure 4(a) demonstrates the overall CPU utilizations of the two tasks during a period of time, in which many individual I/O requests are serviced (as shown in Figure 4(e)). Due to a lower priority, the compute task only runs when the I/O task is idle.

Work-conserving scheduling makes sense under continuous time but can violate I/O prioritization under discontinuous time. As shown in Figure 4(b), neither static nor dynamic priority can be preserved with the 50% CPU cap. For example, if static priority had been enforced, the CPU allocation to the high priority I/O task would not be affected (as shown in Figure 4(c)) and only the low priority CPU-bound task should receive reduced allocation. With dynamic priorities, both the demands of the I/O (i.e., 30% demand) and the compute tasks (i.e., 70% demand) exceed the fair share (i.e., 25%) under the new CPU capacity (i.e., 50%). Thus, CFS should assign equal priorities and allocate a fair share of CPU to each task to achieve max-min fairness, as shown in Figure 4(d).

Running Linux in a VM or container with constrained CPU allocation violates both types of priorities. Figure 4(b) shows the CPU allocation between the two tasks due to WC scheduling. The CPU allocation to the high priority I/O task is significantly lower than those in Figure 4(c) and (d). The problem is that the guest OS assumes dedicated and continuous CPU resources but actually runs on shared, discontinuous CPU allocations. Thus, WC scheduling allows the compute task to consume the CPU that could otherwise be used by the I/O task in the future. Unlike in dedicated systems, where I/O tasks can always timely preempt compute tasks, I/O tasks in a VM with discontinuous CPU are unable to acquire CPU if the VM is not scheduled. As illustrated in Figure 4(e), the period, during which the VM is suspended (dotted bar) due to the cap, prevents half of the I/O requests from being processed. The delay can significantly degrade I/O performance, especially the tail latency shown in Figure 1(d).

4 xBALLOON DESIGN

To enforce static priority, xBalloon guarantees that compute tasks only run when there is slackness in VM CPU allocation. To preserve dynamic priority, xBalloon ensures that the relative priorities of the two tasks faithfully reflect their demands under the constrained CPU allocation. To achieve these goals, xBalloon relies on the use of differential clocks, a CPU balloon process, and a semi-work-conserving scheduling mode. Next, we elaborate on the design of these components in the context of Xen and Linux VMs.

4.1 Differential Clocks

KVM implements two clocks in Linux guests to address inaccurate CPU accounting due to discontinuous time [8]. While `rq_clock` synchronizes with the host clock, `rq_clock_task` only ticks when a VM is running. CFS schedules tasks based on `rq_clock_task` so that vruntimes truly reflect task runtimes. Thus, the short-term priority inversion problem has been addressed by KVM. We port the relative clock in KVM to Xen VMs as `rq_clock_virt` and make it available to other Linux schedulers. Besides preventing short-term priority inversion, the two differential clocks also help enforce static priority between tasks. Note that as shown in Figure 4(c), the low priority task should be the victim of reduced CPU allocation if static priorities are enforced. As will be discussed in § 4.4, the absolute clock `rq_clock` is assigned to the low priority task so that the deprived CPU is accounted to its consumption.

4.2 CPU Balloon

The idea of CPU balloon is inspired by memory ballooning [43], in which the guest OS installs a balloon driver for dynamic memory allocation. The size of the memory balloon represents the amount of memory that has been taken away from the guest. Thus, dynamic memory allocation is realized by inflating and deflating the balloon. Similarly, we use a CPU balloon to represent CPU time the VM voluntarily gives up and reserves for future use.

The CPU balloon acts as a Linux process running in user space. It loops infinitely and at each iteration calls a new system call `sched_balloon` we add to Linux guest to pause the VM. Upon the arrival of an I/O request, the VM is unpaused and resumes normal execution. The balloon process then yields for I/O task execution. As I/O requests wake up the VM, the runtime of the balloon refers to the interval between individual I/O requests. The above actions repeat whenever the balloon process is scheduled.

The purpose of the balloon process is to prevent low priority compute tasks from running when the high priority I/O task is idle, which effectively converts the original work-conserving scheduling to non-work-conserving. However, the balloon should only run under constrained CPU allocation and be disabled if there is CPU resource slack to allow the compute task to run freely. Next, we present the resulting semi-work-conserving scheduling (§ 4.3) and show how to precisely preserve static (§ 4.4) and dynamic (§ 4.5) priorities among tasks.

4.3 Semi-Work-Conserving Scheduling

The goal of semi-work-conserving (SWC) scheduling is to differentiate task scheduling based on the availability of CPU. Under constrained CPU allocation, the VM would be forcibly suspended or de-scheduled if its CPU consumption exceeds the CPU cap or the fair share. To preserve priorities, the guest OS should be in total control of its CPU usage and avoids involuntary suspension and

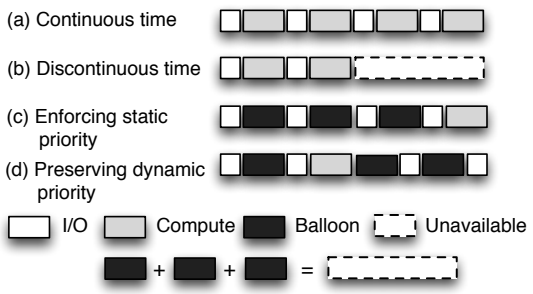

Figure 5: Semi-work-conserving scheduling satisfies resource constraints to avoid involuntary suspension.

descheduling by the hypervisor, while still meeting the resource constraint. As illustrated in Figure 5, to avoid suspension during the black-out period (dotted bar in Figure 5(b)), the VM autonomously schedules the balloon process to reserve CPU for I/O processing. To enforce static priority, as shown in Figure 5(c), the compute task only runs when all I/O requests are serviced. To preserve dynamic priority, as shown in Figure 5(d), the execution of the compute task is interleaved with that of the I/O task. Ideally, the execution time of the balloon, i.e., the reservation of CPU, equals the length of period in which CPU is unavailable to the VM. If so, the VM proactively satisfies the resource constraint and avoids involuntary suspension.

Figure 6 shows how the SWC scheduling realizes such autonomy. During the NWC mode, the balloon is active and throttles the execution of the compute task. The guest OS scheduling switches back to WC mode when the balloon is suspended. The challenge is how to switch between the two modes so that the demand of the I/O task is fully satisfied and the compute task is free to run if there exists CPU slack [2]. Recall that WC scheduling does not violate I/O prioritization under continuous time because the I/O task is always able to preempt the compute task. If we can preserve this property, I/O performance would not be affected by the compute task even under discontinuous time.

Basics of Xen scheduling. In Xen's credit scheduler, CPU allocation is measured by *credits*. As the VM consumes CPU, credits are debited and the balance determines the VM's priority. VMs with non-negative credit balance are assigned with the normal `UNDER` priority while those with negative balance are given a lower `OVER` priority. VMs with the `UNDER` priority take precedence over those in `OVER` in scheduling. In addition, to prioritize I/O-bound VMs, Xen elevates a VM that wakes up from sleep and has non-negative credit balance to the `BOOST` priority, the highest among the three priorities. If a VM's credit usage exceeds the credit cap, it is suspended until it collects sufficient credits. Similarly, using up all of a VM's credit leads to the `OVER` state and the VM will not be able to become `BOOST` or preempt other VMs. Xen refills VMs' credits at the beginning of each accounting period (every 30ms), checks if some VMs exceeds their CPU caps, and re-assigns VM priority based on their credit balances.

[2]We assume a static and strictly higher priority for the I/O task for ease of discussion. With dynamic priorities, the compute task is able to run during the NWC mode even when the balloon is active.

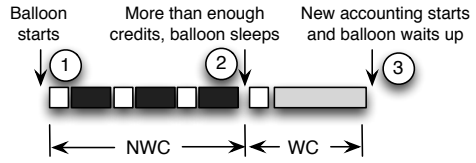

Figure 6: xBalloon switches between non-work-conserving (NWC) and work-conserving (WC) modes based on available credits until next credit refill. The aggregate CPU consumption of I/O (white) and compute (grey) tasks equals the CPU cap or fair share of the VM. Black bar denotes the balloon.

CPU capping. If a VM's CPU usage exceeds its cap, it will be forcibly suspended when periodic accounting is performed. As such, the I/O task will be suspended for a long time (usually a whole 30ms accounting period). The compute task is only allowed to run when there are sufficient credits left until the next accounting period to avoid a forcible suspension. Ideally, SWC allows the guest OS to limit its CPU usage to proactively satisfy the CPU cap.

CPU sharing. While VM suspension due to capping occurs at each accounting, VM preemption can happen any time depending on the co-running VM. To this end, SWC does not intend to prevent a VM from being preempted but guarantees that the compute task does not impede the I/O task preempting other VMs. Specifically, the compute task is allowed to run only if it will not cause the VM to enter the `OVER` state, in which credit balance is negative and a waking VM cannot be boosted.

SWC workflow. As shown in Figure 6, at the beginning of each accounting period, the guest OS is in NWC mode and the balloon process is active (step ①). Each time the balloon is scheduled to run, it calls down to the Linux kernel and checks the current mode from a per-CPU variable `xballoon_mode` shared with the hypervisor. The mode switches to WC once the hypervisor finds that the maximum amount of credits can be debited from the VM until the next credit refill will not cause either a VM suspension due to capping or an entry to the `OVER` state. If so, the balloon suspends itself and waits on a task sleep queue `balloon_queue` (step ②). When the next accounting period starts, the hypervisor notifies the guest OS to wake up the balloon and switch to NWC (step ③).

Robustness of SWC scheduling. We show that SWC does not affect system utilization or application performance: (1) when there is no CPU capping or sharing and the VM has full CPU capacity, VM's credits are always more than enough and the balloon is effectively disabled; (2) If no I/O task is present and CPU allocation is constrained, SWC delays the execution of the compute task during NWC mode and allows it to run at full speed when switching to WC mode. The performance of the compute task is not affected because it receives exactly the capped allocation or the fair share; (3) the balloon is for throttling compute tasks and has a lower priority than the I/O task, thereby not affecting I/O performance if no compute task exists; (4) when multiple VMs, each equipped with SWC, share CPU, it is unlikely that all VMs have negative credits and yield CPU simultaneously. Thus, there will always be one VM running, ensuring that the host machine is work conserving.

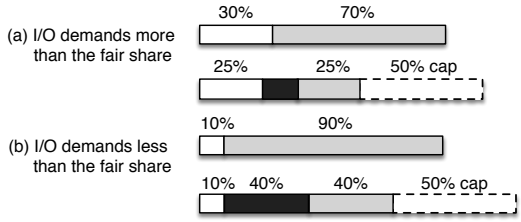

Figure 7: xBalloon helps preserve dynamic priority in CFS under discontinuous time. Black bar denotes the balloon.

4.4 Enforcing Static Priority

If static priority, e.g., real-time vs. best-effort, or kernel-level vs. user-level, is to be enforced, the balloon should strictly prevent the low priority compute task from running unless the I/O demand is fully satisfied.

To this end, xBalloon assigns the absolute clock, i.e., `rq_clock`, to the compute task and the relative clock, i.e., `rq_clock_virt` to the I/O task and the balloon. Since the I/O task has the strictly higher priority, e.g., `SCHED_RR` in Linux, it is always scheduled before the balloon and the compute task. The remaining issue is to ensure the balloon runs before the compute task. xBalloon assigns the normal priority, e.g., `SCHED_OTHER`, to both the balloon and the compute task and uses task runtimes to differentiate scheduling. As the compute task uses the host clock, its runtime is guaranteed to be larger than that of the balloon because the former contains all VM non-running time including the time the VM is paused due to the balloon. As such, the compute task is penalized in scheduling as its "runtime" e.g., vruntime, appears to be quite larger than the balloon. xBalloon can also be extended to support hierarchical priorities with more than two types of tasks. To this end, multiple balloon processes, each with a priority hierarchy, should exist in the guest.

4.5 Preserving Dynamic Priority

The dynamic priority between the I/O and compute tasks can change depending on the demands of individual tasks and the fair share under the constrained CPU allocation. For example, as shown in Figure 7(a), the I/O task, which has a high priority due to smaller runtimes under continuous time, should receive an equal priority as the compute task and a fair share of CPU under the constrained allocation. In contrast, if the I/O task still demands less than the fair share (shown in Figure 7(b)), its demand should be fully satisfied and it is assigned a higher priority.

In Linux CFS, dynamic priorities are determined by vruntimes. As illustrated in Figure 5(a) and (b), I/O requests (white bars) arrive at discrete time and the real I/O demand, i.e., the aggregate of all white bars under continuous time, cannot be fully presented to CFS for scheduling under discontinuous time, thereby violating the dynamic priority. To address this issue, we integrate the balloon process into CFS scheduling and extend the CFS fair sharing to include CPU reservations for future use (i.e., the balloon). As shown in Figure 5(c), if the balloon (the black bar) is properly scheduled, the I/O demand can be fully exposed to CFS. Next, we discuss how the balloon helps preserve dynamic priority for the two cases shown in Figure 7 using extended fair sharing.

Fully satisfying I/O demand. In extended fair sharing, the balloon, the I/O and compute tasks are scheduled by CFS as regular processes, i.e., with the `SCHED_OTHER` policy and all use the relative clock `rq_clock_virt`. Note that the balloon's runtime also includes the time the VM is paused by the balloon. As discussed in § 4.2 the demand of the balloon (r_{bal}) is the inverse of the I/O demand [3]($r_{io} = 1 - r_{bal}$). Assume there are n processes, including the I/O task, sharing the CPU and the VM CPU allocation is c. The fair share of the CPU allocation among n tasks is $\frac{c}{n}$. For compute-bound tasks, their demand/runtime (r_{comp}) will be bounded by the fair share ($\frac{c}{n}$). If the I/O tasks demands less than the fair share, as shown in Figure 7(b), $r_{io} < \frac{c}{n}$ and given that $c \leq 1$, we have the following strict order between task runtimes if they are scheduled by CFS: $r_{bal} > r_{comp} > r_{io}$. Thus, the I/O task is guaranteed to have the smallest vruntime and its demand will be fully satisfied.

Enforcing fair sharing. When the I/O demand is larger than the fair share (as shown in Figure 7(a)), a **tweak** is needed to guarantee that CFS fairly allocates CPU to the I/O task. CFS clamps a waking I/O task's vruntime to the minimum vruntime in the runqueue minus an offset to prevent I/O tasks from monopolizing CPU. This design works effectively and correctly under continuous time as I/O tasks that demand more than the fair share are guaranteed not to have too small vruntimes compared to the runqueue minimum. However, this property does not hold under discontinuous time, where the demand of the I/O task exceeds the fair share because the fair share itself drops. Since CFS is not aware of time discontinuity, it frequently resets the vruntime of the I/O task but not that of the compute task, making it impossible for the I/O task to achieve the fair share. To this end, we make a minimal change to CFS: when the balloon is running and CFS experiences discontinuous time, the I/O task is allowed to preserve its vruntime when waking up.

5 IMPLEMENTATION

We implemented xBalloon in Linux 3.18.21 and Xen 4.5.0. To support two differential clocks in guests, we ported the `rq_clock_task` in KVM guest to Xen VM. To support VM pause, we added a new system call (`sched_balloon`) in Linux kernel and a new hypercall (`SCHEDOP_sleep`) in Xen. xBalloon used the existing VM pause interface in Xen, but unlike the existing `vcpu_pause` holding a per-vCPU lock, we implemented a fast path `vcpu_fast_pause` for handling frequent VM sleeps. Since VM pause is to reserve CPU for future I/O, it should be canceled if an I/O request is already pending to avoid delayed I/O processing. We added this optimization in both Xen and Linux to avoid problems such as transmission delay of TCP ACKs to senders [45].

To support SWC scheduling, we defined work-conserving and non-work-conserving mode as a per CPU variable in shared memory. We added a new virtual interrupt (VIRQ) to guest OS to notify the change of xBalloon mode. The corresponding interrupt handler in the guest then queries xBalloon mode using another hypercall `query_xballoon_mode`. Last, we allowed the VM to be woken up from xBalloon sleep by timer interrupts to improve the VM's responsiveness in case of no I/O coming for a long period of time. We set the frequency of timer interrupts in the guest to 1000HZ.

[3]We assume the total CPU capacity to be 1 and the demands and allocation are in percentage.

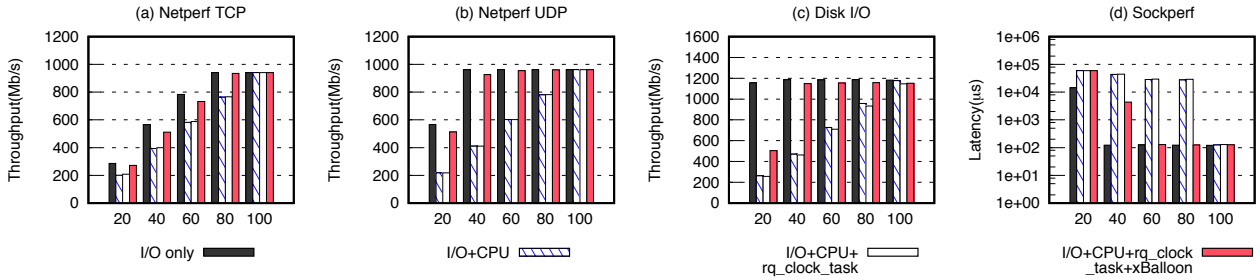

Figure 8: xBalloon improves I/O performance by enforcing static priority. The x-axis shows CPU caps.

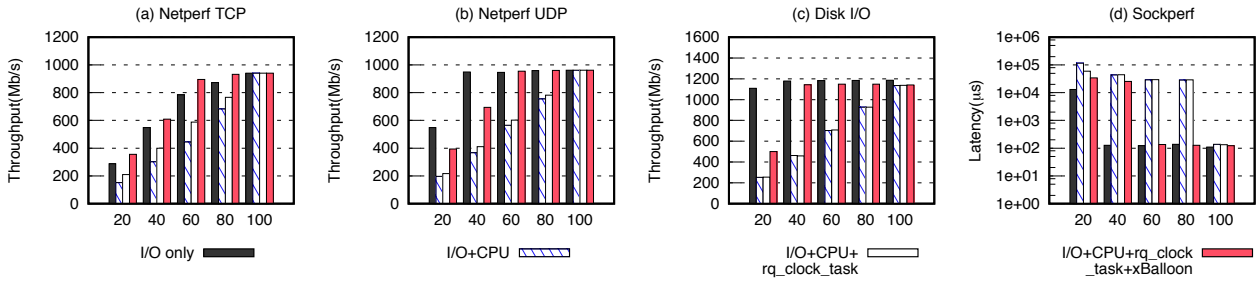

Figure 9: xBalloon improves I/O performance by preserving dynamic priority in CFS. The x-axis shows CPU caps.

6 EVALUATION

In this section, we evaluate xBalloon using both micro-benchmarks and real-world applications. We first created a controlled environment, where a one-vCPU VM was used and CPU discontinuity was due to CPU capping, to study the effectiveness of xBalloon in improving I/O performance and preserving priorities (§ 6.1). We then show results on the CPU sharing case (§ 6.2) and evaluate xBalloon with two 4-vCPU VMs running realistic workloads (§ 6.3). Finally, we discuss xBalloon's overhead and its applicability to different workloads (§ 6.4), and present a study on Amazon EC2 (§ 6.5).

Experiments setup. Our experiments were performed on two DELL PowerEdge T420 servers, connected by Gigabit Ethernet. Each server was equipped with two six-core Intel Xeon E5-2410 1.9GHz processors, 32GB memory, one Gigabit Network card and a 1TB 7200RPM SATA hard disk. We ran Linux 3.18.21 as the guest and dom0 OS, and Xen 4.5.0 as the hypervisor. The VMs were configured with one vCPU and 4GB memory.

6.1 CPU Capping

Capping provides a convenient means to control resource allocation and evaluate xBalloon at various levels of CPU allocation.

6.1.1 Improving I/O Performance. We begin with micro benchmarks `netperf`, `sockperf` and a synthetic disk benchmark that sequentially accesses a 2GB file to measure TCP/UDP throughput, network tail latency and disk read throughput, respectively. The compute task was a `while(1)` CPU hog, which had almost zero memory footprint. Figure 8 and Figure 9 show the performance of different I/O workloads under distinct CPU caps due to static and dynamic priorities. The results shown were the average of ten runs. A cap of 100 refers to a full pCPU capacity. In general, for the `I/O only` case, I/O performance initially stayed unaffected as the cap dropped until the cap fell below I/O's CPU demand. For example, I/O performance dropped at cap 60 for the `netperf` TCP test.

I/O Performance due to static priority. Figure 8 shows I/O performance while the I/O task was set to real-time priority. It suggests that I/O suffered significant performance loss when co-located with the CPU-bound task even with a strictly higher priority (denoted as `I/O+CPU`). I/O had as much as 27.4%, 57.2%, and 77% throughput loss, and 417x latency hike for netperf TCP, UDP, disk I/O, and sockperf, respectively. As scheduling was not based on the vruntimes of the two tasks under different priorities, no short-term priority inversion happened and the performance loss was due to long term priority inversions. Therefore, `rq_clock_task` did not help in I/O performance. In contrast, xBalloon achieved near-native performance for all four I/O workloads compared to case `I/O only`. As CPU cap approached to the I/O demand or fell below it, e.g., TCP throughput under cap 60 in Figure 8(a) or disk throughput under cap 20 in Figure 8(c), xBalloon incurred performance drop compared to the reference performance. This is due to the balloon's CPU consumption, which together with I/O's CPU demand, goes beyond the CPU cap. Further, our results also show that xBalloon significantly reduces the variation of sockperf latency compared to that under `I/O+CPU`.

I/O Performance due to dynamic priority. Figure 9 shows the results on fair sharing enforced by CFS dynamic priorities. All tasks including the balloon were assigned the same policy `SCHED_OTHER` and scheduled by Linux CFS. Compared to Figure 8, I/O performance with xBalloon was worse than the reference I/O performance due to fair sharing CPU resources among tasks, e.g. UDP throughput under cap 40 in Figure 9(b). Another observation in Figure 9 is that

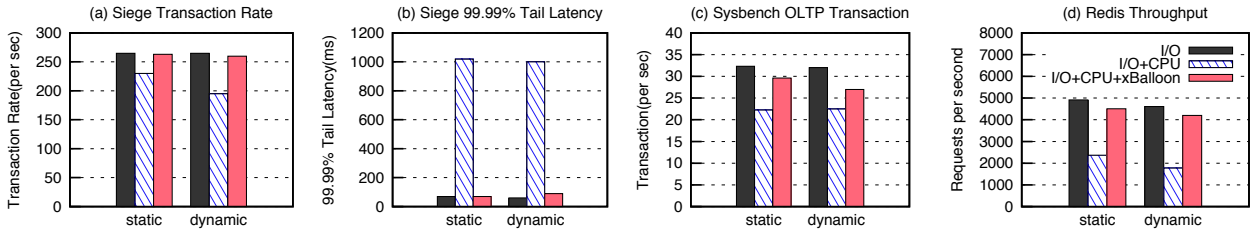

Figure 10: xBalloon improves the performance of Web server and database applications under constrained CPU allocation.

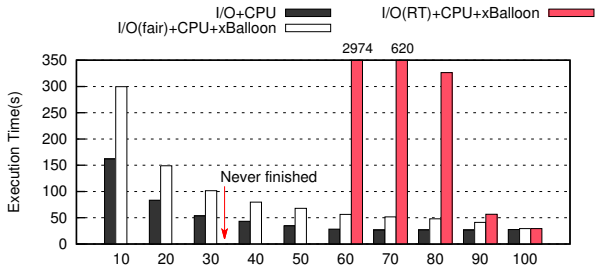

Figure 11: xBalloon deprives the compute task of CPU when the cap is lower than the I/O demand and allows the compute task to use slack CPU resources when CPU cap increases.

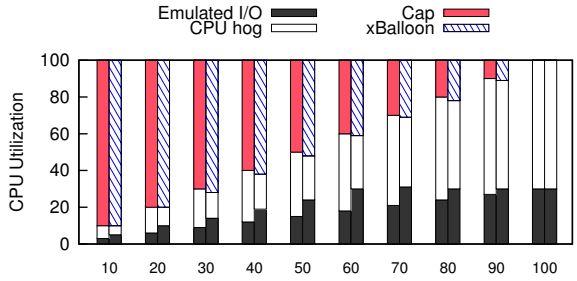

Figure 12: xBalloon helps CFS satisfy I/O's full demand and later enforce fair share between I/O and compute as CPU allocation drops. The x-axis shows CPU caps.

`rq_clock_task` helped in I/O performance in some cases. For example, for `netperf` TCP in Figure 9(a), it improved throughput over `I/O+CPU` by as much as 12%. The improvement was due to the avoidance of short-term priority inversions. However, `rq_clock_task` was not effective for performance loss due to long-term priority inversions. In contrast, xBalloon improved I/O performance over `rq_clock_task` on average by 48.5%, 57.4%, 95.3%, and 125x for netperf TCP, UDP, disk I/O, and sockperf latency, respectively.

Note that xBalloon even outperformed `I/O only` in some tests, e.g., TCP throughput under cap 40 in Figure 9(a). Similar results can also be observed in the experiments in which multiple VMs CPU share the CPU. The reason will be revealed in § 6.2.1.

6.1.2 Application Results. Next, we evaluate xBalloon using real world I/O-intensive applications. If not otherwise stated, the compute task was `mcf` from the SPECCPU 2006 benchmarks [15] and the VM had a cap of 50.

Web server applications. We hosted a `Nginx` [11] HTTP server in the VM and ran `Siege` [13] on another machine as the client. We simulated 200 users that sent a total of 4,000 requests. As shown in Figure 10(a) and (b), Nginx suffered significant performance loss when co-running with the compute-bound program. In contrast, xBalloon improved the throughput and 99^{th} percentile latency by 14.4%, 14x under static priority, and by 33.3%, 10x under dynamic priority, respectively.

Database applications. First, we evaluated the performance of `Sysbench` [16], an OLTP application benchmark running on a MySQL database. We created 10 MySQL tables and each table contained 100,000 records. A client which contained 100 threads performed OLTP transactions in the database. As shown in Figure 10(c), xBalloon increased Sysbench transaction rate by 32.9% under static priority and 20% under dynamic priority in comparison with `I/O+CPU`, respectively. Second, we tested NoSQL database applications. We ran `Redis` [12] 3.0.7 as a server and used redis-benchmark to send 100,000 requests from a client. As shown in Figure 10(d), xBalloon achieved close performance to `I/O only` and outperformed `I/O+CPU` by 90.2% under static priority and 136% under dynamic priority, respectively.

6.1.3 Preserving Static and Dynamic Priorities. It is challenging to directly verify priority preservation. The compute task's CPU usage reported by Linux may be inaccurate when using the global clock `rq_clock`. It can contain the time the VM is paused by xBalloon . Instead, we used two indirect approaches.

Enforcing static priority. We changed the CPU hog to loop for a specified number of iterations and used its completion time to infer its CPU allocation. Figure 11 shows its completion time under various CPU caps. The completion time in `I/O+CPU` is the baseline, in which the CPU hog acquired excessive CPU due to work-conserving scheduling in the guest. When fair sharing was enabled (white bar), the completion time of the compute task increased, indicating that the compute task was allocated less CPU. When static priority was enforced (red bar), the compute tasks failed to complete under low CPU caps (i.e., the missing data pointed by the arrow in Figure 11). xBalloon enforced strictly higher priority for the I/O task and starved the CPU task when the cap cannot fully satisfy I/O's demand. As cap increased, the CPU task was able to use slack CPU and complete, but with longer completion time compared to that in fair sharing.

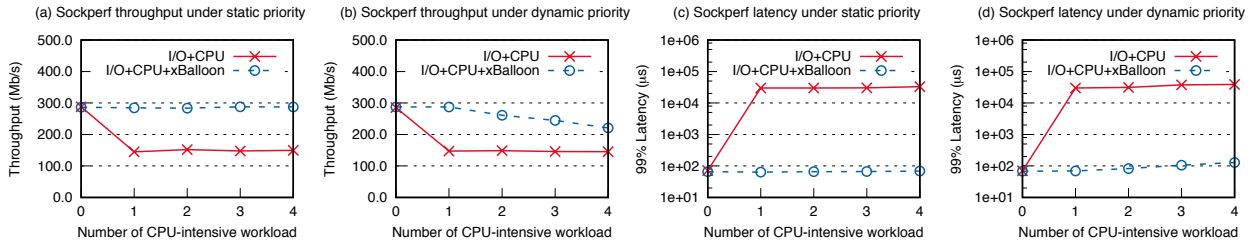

Figure 13: I/O performance with a varying number of co-located CPU workloads.

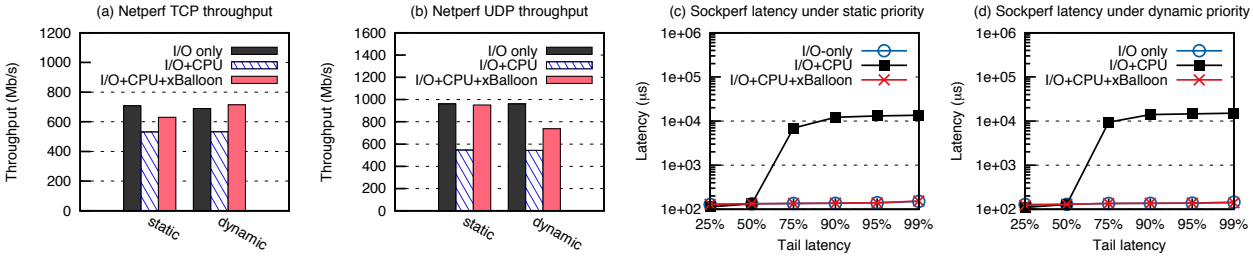

Figure 14: xBalloon improves I/O performance with one I/O-bound VM and one CPU-bound VM sharing the CPU.

Preserving dynamic priority. Although Figure 11 qualitatively demonstrates the preservation of static priority, it cannot quantitatively verify that the two tasks were indeed fairly scheduled under dynamic priority. I/O processing in the guest kernel including that in the soft and hard interrupt contexts is charged to the CPU task if it happens to be the one running when I/O requests arrive. To this end, we emulated the I/O task's CPU demand using a user-level process, which indefinitely repeats the cycle of computing and idling, whose ratio determines the CPU demand. We set the emulated I/O task to demand 30% of CPU. Since the process does not incur kernel processing, we use it to study how xBalloon helps preserve dynamic priority.

Figure 12 shows the CPU allocations with (right bar) and without (left bar) xBalloon under various cap settings. From the left bars in each group, we can see that the I/O task can only take 30% of the CPU time left by the cap no matter what was the new fair share based on the cap value. In contrast, with xBalloon , CFS included future CPU allocations (i.e., the balloon) in fair sharing. CFS effectively satisfied the I/O demand in full when the cap was large and enforced the fair share between the two tasks as CPU cap dropped. We conclude that xBalloon preserves dynamic priority by initially granting the I/O task a higher priority and smoothly demoting it to an equal priority.

Varying number of CPU workloads. In this experiment, we co-locate an I/O task with a varying number of CPU workloads. We measured TCP throughput and the 99^{th} tail latency using `sockperf`. Figure 13(a)-(d) show I/O performance when the `sockper` server process ran with one to four CPU hogs. Under static priority, as shown in Figure 13(a) and (c), xBalloon was always able to prioritize the I/O task. TCP throughput and latency were consistent despite the increasing levels of contention from co-located compute tasks. Under dynamic priority, the relative importance of the I/O task and the compute tasks are determined by their CPU usage. As shown in Figure 13(b) and (d), I/O throughput gradually dropped and latency increased as the number of CPU hogs increased. When there were only two tasks sharing the CPU, i.e., the I/O task and one compute task, the CPU demand of the `sockperf` server process is lower than the fair share, i.e., 50% of CPU. Thus, xBalloon prioritized the I/O task and achieved similar performance compared to that under static priority. As CPU competition ramped up, the CPU allocation to each task decreased and the demand of the I/O task gradually exceeded the fair share. To enforce max-min fairness, xBalloon assigned an equal priority to the I/O task. The degraded I/O performance was due to a decreasing fair share and CPU allocation to the I/O task. Nevertheless, xBalloon substantially improved I/O performance under both static and dynamic priorities compared to the `I/O+CPU` case.

6.2 CPU Sharing

CPU sharing presents a more challenging consolidation scenario, in which multiple VMs share the same pCPU. We evaluate xBalloon with two VMs sharing one pCPU but the results can be extended to more than two VMs.

6.2.1 Sharing with CPU-bound VM. We start with a relatively simple scenario, in which one VM ran a mix of I/O- and CPU-bound workloads and the other VM was CPU-bound. Both VMs were assigned with equal weights. As the CPU-bound VM was unable to preempt the I/O-bound VM, xBalloon should guarantee that the I/O task, even co-running with the compute task, can preempt the CPU-bound VM at any time.

Figure 14(a) and (b) show the `netperf` TCP/UDP throughput under static and dynamic priorities. xBalloon improved TCP and

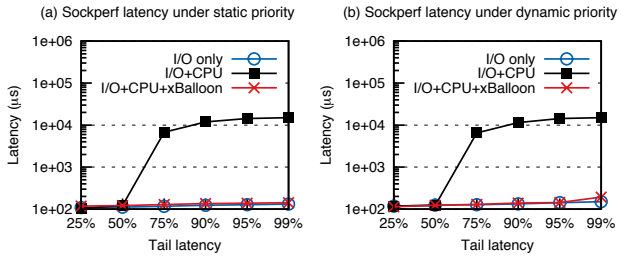

Figure 15: xBalloon improves I/O performance with two I/O-bound VMs sharing the CPU.

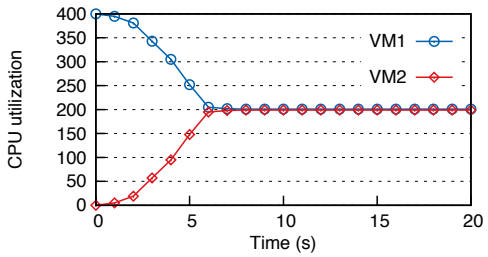

Figure 16: xBalloon does not cause significant idleness in the host and preserves fairness in CPU allocation.

UDP throughput over `I/O+CPU`. The improvement on TCP throughput was 18.6% and 34.4% under static and dynamic priority, respectively. Similar to our observations in Figure 9(a) under CPU capping, xBalloon with dynamic priority outperformed the reference I/O performance in `I/O only`. The reason is that when the I/O task ran alone or ran in a real-time priority with xBalloon , it used as much time as it can until the entire VM is suspended due to CPU cap or descheduled due to insufficient credits. The VM was suspended for one accounting period (30ms in Xen) or descheduled for one time slice (also 30ms in Xen) before next credit refill.

In contrast, when the I/O task ran with xBalloon under CFS fair sharing, it did not fully exercise its CPU demand. The VM can maintain a utilization no larger than the CPU cap or never used up its credit so as to enter the `OVER` state, either of which avoided long time freeze of the VM. This finding suggests that temporarily throttling I/O demand under constrained CPU allocation can lead to superior performance. As shown in Figure 14(c) and (d), xBalloon achieved almost identical latency distribution compared to the reference latency distribution while `I/O+CPU` had wildly growing 75^{th} percentile latency.

6.2.2 Sharing between I/O-bound VMs. The most challenging scenario is to consolidate two VMs, each running a mix of I/O- and CPU-bound workloads, and to preserve I/O prioritization in each VM. As the two VMs shared the Gigabit NIC on the host, the network would be the bottleneck if throughput tests were performed. Thus, we measured the tail latency on each VM using `sockperf` UDP test. The CPU-bound tasks were `mcf`. Figure 15(a) and (b) show the latency distribution of the two VMs. With xBalloon , both VMs had predictable and low tail latency close to the reference `I/O only` case. The latencies were not affected by co-running compute tasks, showing a clear preservation of I/O prioritization in both VMs.

6.3 Results on SMP VMs

This section presents an evaluation of xBalloon with SMP VMs. We are interested in evaluating the effectiveness of xBalloon in preserving I/O prioritization for multi-threaded workloads and studying its impact to the fairness of SMP CPU allocation and the overall system utilization. We configured two 4-vCPU VMs to share a set of 4 pCPUs and pinned each vCPU in a VM to a separate pCPU. As a result, two vCPUs from different VMs compete for the same pCPU in a time-sharing manner. We used the multi-threaded `Data Caching` benchmark from Cloudsuite [4] as the latency-sensitive, I/O-bound workload and `mcf` as the compute-bound workload. The Data Caching benchmark uses `memcached` [9] configured with 4 threads to simulate the behavior of a Twitter caching server using the twitter dataset. Two client machines, each with 4 worker threads and 20 connections were used to send requests at a rate of 5000 per second. The ratio of `GET` and `SET` was set to 4:1. To fully utilize four vCPUs, four copies of `mcf` were run in each VM.

Since the caching benchmark is network intensive, the 1Gbps Ethernet on the host machine was the bottleneck when two VMs both ran `memcached`. Thus, We measured the performance of multi-threaded I/O in one VM while the other VM only executed the compute-bound programs. Table 1 shows the throughput (request per second) and latency (ms) of `memcached`. `I/O only` was the case when one VM was dedicated to `memcached` and the other VM hosted the compute-bound workload. Because Xen prioritizes I/O workloads at the hypervisor level, the consolidation of I/O- and CPU-bound workloads in separate VMs does not affect I/O performance, we regard `I/O only` as the reference I/O performance. Similar to previous observations, the co-location of `memcached` and `mcf` (denoted as `I/O+CPU`), in addition to sharing pCPUs with another compute-bound VM, inflicted significant I/O performance degradation. While throughput was not much affected, the average and tail latency were drastically increased by several orders of multitude under both static and dynamic priority. To preserve I/O prioritization, we enabled four balloon processes, each bound to a vCPU, in the `memcached` VM. Results (in bold font) show that xBalloon effectively preserved both static and dynamic priorities and achieved performance close to the reference performance.

Figure 16 shows the CPU utilizations of the two VMs when they were both running a mixture of `memcached` and `mcf` with xBalloon enabled in both VMs. Recall that xBalloon switches to NWC mode at the beginning of each accounting period. Thus, it is possible that both VMs could schedule the balloon process and yield CPU at the same time, causing idleness in the host machine. Figure 16 show that it took approximately 6s for the CPU allocation to converge to the fair share (i.e., 200%, half of the 4-pCPU capacity) and no significant CPU idling was observed in the host. As discussed in § 4.3, it is unlikely that both VMs yield CPU. Even if this happens, the host only experiences transient idleness and xBalloon helps to coordinate the scheduling of the two VMs such that their busy and idle (due to the balloon) periods are interleaved.

6.4 Discussion

Overhead. xBalloon's overhead includes the time spent in user space, in the guest kernel, and inside Xen. As the balloon is woken

Experiment	rps	avg_lat	90^{th}	95^{th}	99^{th}	std
I/O only	5010.3	0.167	0.192	0.204	0.251	0.094
I/O+CPU(s)	5015.6	38.053	79.311	80.393	81.152	31.418
I/O+CPU+xBalloon(s)	4997.7	**0.169**	**0.197**	**0.211**	**0.263**	**0.097**
I/O+CPU(d)	5023.1	39.352	80.128	82.336	83.713	32.926
I/O+CPU+xBalloon(d)	4998.3	**0.164**	**0.190**	**0.204**	**0.259**	**0.114**

Table 1: The throughput (request per second) and latency (ms) of the caching benchmark in various test scenarios. *S* and *D* denote static and dynamic priority for the I/O workload, respectively.

up by I/O events, the frequency of xBalloon invocation depends on the intensity of the I/O workload. The overhead is negligible in most cases. As CPU cap or share drops close to the actual I/O demand, xBalloon can lower the CPU available to serving I/O, leading to moderate degradation of I/O performance.

Target I/O workloads. For most I/O-bound applications, xBalloon can effectively improve its throughput or latency when co-locating with CPU-intensive applications under discontinuous time. However, xBalloon is not effective for workloads with both substantial I/O and computational requirements. The reason is that the priority inversion issue is not significant between compute-intensive I/O workloads and real compute-bound workloads, thereby leaving little room for improvement. xBalloon is also effective in enforcing priority between I/O-bound workloads and compute-intensive workloads that use the CPU less intensively. For example, some compute-intensive programs have interleaved busy and idle periods. If an I/O task is set to a static and higher priority, xBalloon strictly prioritizes the I/O task. If dynamic priority is to be preserved, the relative priority between an I/O task and a compute task depends on their CPU usage. If a "compute-intensive" workload uses less CPU than the I/O task, xBalloon would prioritize the former and whichever task that is deemed to have a higher priority under CFS.

Performance penalty to CPU-bound workloads. Work conserving scheduling allows CPU-bound workloads to gain excessive CPU resources due to priority inversion and causes significant delays to I/O processing. Therefore, if priority is preserved, CPU-bound workloads would inevitably experience performance slowdowns. As discussed in 6.1.3, CPU-bound programs could suffer from starvation if static priority is enforced and an I/O-bound task could use up all CPU allocations. Under dynamic priority, xBalloon ensures max-min fairness among tasks, which can leads to reduced CPU allocation to CPU-bound tasks. As shown in Figure 11, the CPU-bound task was slowed down by up to 48%. We believe that performance penalty to CPU-bound tasks is necessary for guaranteeing I/O performance if the CPU capacity of the VM cannot satisfy the aggregate demand of I/O and compute tasks.

Applicability to other hypervisors. xBalloon can be extended to other hypervisors, such as KVM. The changes to hypervisor are new mechanisms to support efficient VM pause and the notification of the switch of NWC and WC modes. No algorithmic change is needed to the VM scheduling. To port xBalloon to KVM, the key is to define the SWC mode in the context of CFS, the VM scheduler in KVM, instead of Xen's credit scheduler.

6.5 Results on Amazon EC2

Last, we show the effectiveness of xBalloon on Amazon EC2. Even though we do not have access to the hypervisor, xBalloon can still pause an EC2 instance. We used the hypercall `SCHEDOP_block` to pause the VM and disabled SWC scheduling. We built a Amazon Machine Image (AMI) with our modified Linux kernel on an `m3.medium` instance, whose network bandwidth is capped at 340 Mbps and CPU at about 60%. Thus, we can only test on network latency. Figure 17 shows the results on TCP 99^{th} percentile latency using `sockperf` and the average latency of `ping`. Since Linux network stack directly responds to `ping`'s ICMP requests, it is not possible to assign real-time priority to a user space process. Thus, we skipped real-time results for `ping`. The figure shows that xBalloon effectively reduced TCP tail latency by 68% and even achieved a lower latency in `ping` test than the I/O only case. Although the test lacks important optimizations for xBalloon , it shows that wiser scheduling inside the guest, without undermining the autonomy of the VM or changing resource management at the hypervisor, can greatly improve the I/O performance.

7 RELATED WORK

Bridging the semantic gaps. The semantic gap in virtualized environments has been a well-recognized issue. There were studies on bridging the gaps for VM introspection [20, 21, 38], adapting TCP congestion control [19], optimizing virtualized I/O scheduling [30, 36], inferring information about process [27] and buffer management [26] inside VMs; however they focused on exposing VM information to the hypervisor to aid bare-metal resource management. Such designs not only make the hypervisor more complex but also make enforcing fairness between tenants difficult. In contrast, xBalloon focuses on exposing the information on resource allocation at the hypervisor to the guest OS to make wise scheduling decisions inside VMs. xBalloon improves I/O performance in VMs without violating the autonomy VM's resource management or seeking any algorithmic changes at the hypervisor for I/O performance optimization.

Optimizing the critical I/O path. As virtualization introduces additional layers of abstraction, indirection, and data movement, the critical I/O path in virtualized systems contains excessive asynchrony and latency [40]. To reduce virtualization overhead, existing work optimizes the critical I/O path by offloading performance critical operations from VMs to the hypervisor, such as TCP ACK generation and congestion control [22, 29], TCP segmentation/checksum calculation [33], and device driver functionalities [34], or by packet

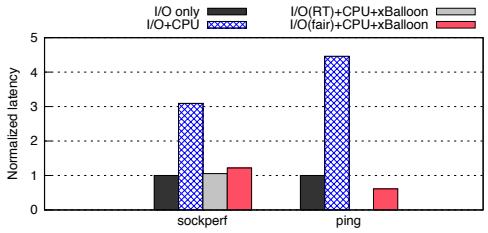

Figure 17: xBalloon helps reduce network latency on EC2.

coalescing [35]. Other work proposed more aggressive changes to I/O virtualization via exit-less interrupt delivery [24] and I/O path reconstruction [23, 25, 32]. xBalloon is complementary to these approaches and guarantees critical I/O operations to be scheduled timely in the guest OS.

Reducing VM scheduling delays. The VM scheduling delay prevents I/O operations from VMs being timely processed. To address this issue, many efforts, like shortening time slices [17, 46], partial boosting [30, 39] or dedicating CPUs [45] are proposed. There are also other work focusing on optimizing scheduling for MapReduce clusters [28], data center workloads [47] and I/O interrupt handling on SMP VMs [19]. We accept that long scheduling delays may be inevitable due to resource sharing, but I/O performance can be optimized given the resource constraint. We show that long I/O processing delay can be avoided by prioritizing I/O in guest scheduling. VM scheduling delay can also be alleviated by using smaller time slices in the hypervisor. There are two drawbacks to use small time slices. First, while I/O workloads benefit from short time slices due to more responsive scheduling, CPU-bound workloads could suffer performance degradation because frequent context switching causes loss of data locality. Second, the change of VM time slice at the hypervisor will affect all users. It may be undesirable for some users. In contrast, xBalloon allows the guest OS to autonomously decide how to use its CPU. Our work is most related to task-aware VM scheduling [30], which prioritizes VMs doing I/O and de-schedules them once the hypervisor detects the VMs are performing computation. xBalloon offers several advantages over this approach. First, it precisely preserves the static and dynamic priorities. Second, xBalloon does not make *algorithmic changes* to a hypervisor's core scheduling algorithm.

8 CONCLUSION

This paper demonstrates that task scheduling in the guest OS should be adapted to efficiently utilize virtual CPU resources. Time discontinuity due to CPU multiplexing or capping can render I/O prioritization in the guest ineffective, leading to I/O performance loss and unpredictability. This paper presents xBalloon , a lightweight approach to preserving static and dynamic priorities between I/O- and compute-bound tasks. xBalloon centers on two designs: a CPU balloon representing CPU reservations and semi-work-conserving scheduling in VM. We demonstrate that xBalloon is effective in boosting I/O performance and preserving priorities in both CPU capping and sharing.

ACKNOWLEDGMENTS

We are grateful to our reviewers for their comments on this paper and our shepherd Ji-Yong Shin for his suggestions. This research was supported in part by U.S. National Science Foundation grants CNS-1649502, IIS-1633753 and by Hong Kong Research Grants Council Collaborative Research Fund under grant C7036-15G.

REFERENCES

[1] Amazon ec2 t2 instances. https://goo.gl/vbCpkV.
[2] Amazon elastic compute cloud(ec2). http://aws.amazon.com/ec2/.
[3] Aws instance usage report from rightscale users. https://goo.gl/AjDKiK.
[4] Cloudsuite: The benchmark suite of cloud services. http://cloudsuite.ch/.
[5] Credit scheduler cap. https://wiki.xen.org/wiki/Credit_Scheduler#Cap.
[6] The docker container. https://www.docker.com/.
[7] Kernel based virtual machine. http://www.linux-kvm.org/.
[8] Kvm cpu accounting. https://github.com/penberg/linux-kvm/blob/master/kernel/sched/sched.h.
[9] Memcached. https://memcached.org/.
[10] Netperf. http://www.netperf.org/.
[11] Nginx. https://www.nginx.com/.
[12] Redis. http://redis.io.
[13] Siege. https://www.joedog.org/siege-home/.
[14] Sockperf. https://github.com/Mellanox/sockperf.
[15] Spec cpu 2006 benchmark. https://www.spec.org/cpu2006/.
[16] The sysbench benchmark suite. https://github.com/akopytov/sysbench.
[17] Ahn, J., Park, C. H., and Huh, J. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2014).
[18] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T. L., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2003).
[19] Cheng, L., Wang, C., and Lau, F. C. M. PVTCP: towards practical and effective congestion control in virtualized datacenters. In *Proc. of IEEE International Conference on Network Protocols (ICNP)* (2013).
[20] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy(SP)* (2011).
[21] Fu, Y., and Lin, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy(SP)* (2012).
[22] Gamage, S., Kangarlou, A., Kompella, R. R., and Xu, D. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)* (2011).
[23] Gamage, S., Xu, C., Kompella, R. R., and Xu, D. vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2014).
[24] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., and Tsafrir, D. Eli: bare-metal performance for i/o virtualization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).
[25] Hwang, J., Ramakrishnan, K., and Wood, T. Netvm: high performance and flexible networking using virtualization on commodity platforms. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2014).
[26] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006).
[27] Jones, S. T., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., et al. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference(USENIX ATC)* (2006).
[28] Kang, H., Chen, Y., Wong, J. L., Sion, R., and Wu, J. Enhancement of xen's scheduler for mapreduce workloads. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC)* (2011).
[29] Kangarlou, A., Gamage, S., Kompella, R. R., and Xu, D. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).
[30] Kim, H., Lim, H., Jeong, J., Jo, H., and Lee, J. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)* (2009).
[31] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference*

on Computer Systems (EuroSys) (2014).

[32] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2014).

[33] Menon, A., Cox, A. L., and Zwaenepoel, W. Optimizing network virtualization in xen. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)* (2006).

[34] Menon, A., Schubert, S., and Zwaenepoel, W. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009).

[35] Menon, A., and Zwaenepoel, W. Optimizing tcp receive performance. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)* (2008).

[36] Ongaro, D., Cox, A. L., and Rixner, S. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments(VEE)* (2008).

[37] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (1974).

[38] Saberi, A., Fu, Y., and Lin, Z. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)* (2014).

[39] Shim, H., and Lee, S.-M. Cfs-v: I/o demand-driven vm scheduler in kvm. *Software R&D Center, Samsung Electronics* (2014).

[40] Suo, K., Rao, J., Cheng, L., and C. M. Lau, F. Time capsule: Tracing packet latency across different layers in virtualized systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)* (2016).

[41] VMware. Vmware horizon view architecture planning 6.0. In *VMware Technical White Paper* (2014).

[42] Waldspurger, C., and Rosenblum, M. I/O virtualization. *Commun. ACM 55*, 1 (Jan. 2012).

[43] Waldspurger, C. A. Memory resource management in vmware ESX server. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)* (2002).

[44] Wang, G., and Ng, T. S. E. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of the 29th Conference on Information Communications (INFOCOM)* (2010).

[45] Xu, C., Gamage, S., Lu, H., Kompella, R., and Xu, D. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)* (2013).

[46] Xu, C., Gamage, S., Rao, P. N., Kangarlou, A., Kompella, R. R., and Xu, D. vSlicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the international symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2012).

[47] Xu, Y., Bailey, M., Noble, B., and Jahanian, F. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)* (2013).