Effectively Mitigating I/O Inactivity in vCPU Scheduling

Weiwei Jia[1][2], *Cheng Wang*[1], *Xusheng Chen*[1], *Jianchen Shan*[2], *Xiaowei Shang*[2]
Heming Cui[1], *Xiaoning Ding*[2], *Luwei Cheng*[3], *Francis C. M. Lau*[1], *Yuexuan Wang*[1], *Yuangang Wang*[4]
[1] *University of HongKong* [2] *New Jersey Institute of Technology* [3] *Facebook* [4] *Huawei*

Abstract

In clouds where CPU cores are time-shared by virtual CPUs (vCPU), vCPUs are scheduled and descheduled by the virtual machine monitor (VMM) periodically. In each virtual machine (VM), when its vCPUs running I/O bound tasks are descheduled, no I/O requests can be made until the vCPUs are rescheduled. These inactivity periods of I/O tasks cause severe performance issues, one of them being the utilization of I/O resources in the guest OS tends to be low during I/O inactivity periods. Worse, the I/O scheduler in the host OS could suffer from low performance because the I/O scheduler assumes that I/O tasks make I/O requests constantly. Fairness among the VMs within a host can also be at stake. Existing works typically would adjust the time slices of vCPUs running I/O tasks, but vCPUs are still descheduled frequently and cause I/O inactivity.

Our idea is that since each VM often has active vCPUs, we can migrate I/O tasks to active vCPUs, thus mitigating the I/O inactivity periods and maintaining the fairness. We present VMIGRATER, which runs in the user level of each VM. It incorporates new mechanisms to efficiently monitor active vCPUs and to accurately detect I/O bound tasks. Evaluation on diverse real-world applications shows that VMIGRATER can improve I/O performance by up to 4.42X compared with default Linux KVM. VMIGRATER can also improve I/O performance by 1.84X to 3.64X compared with two related systems.

1 Introduction

To ease management and save energy in clouds, multiple VMs are often consolidated on a physical host. In each VM, multiple vCPUs often time-share a physical CPU core (aka. pCPU). The VMM controls the sharing by scheduling and descheduling the vCPUs periodically. When a vCPU is scheduled, tasks running on it become active and make progress. When a vCPU depletes its time slice, it is descheduled, and tasks on it become inactive and stop making progress.

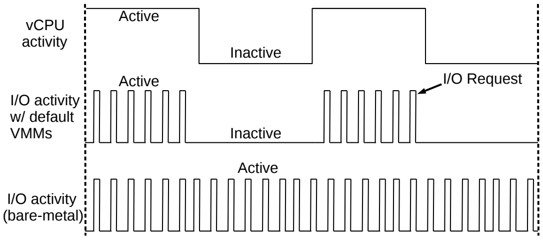


Figure 1: ***I/O inactivity*.**

vCPU inactivity leads to a severe I/O inactivity problem. After the vCPU is descheduled, the I/O tasks on it become inactive and cannot generate I/O requests, as shown in the first two curves in Figure 1. The inactive periods can be much longer than the latencies of storage devices. Typical time slices can be tens of milliseconds; the storage device latencies are a few milliseconds for HDDs and microseconds for SSDs. Thus, during the I/O inactive periods, I/O devices (both physical and virtual devices) may be under-utilized. The under-utilization becomes more serious with a higher consolidation rate (i.e., the number of vCPUs shared on each pCPU), because a vCPU may need to wait for multiple time slices before being rescheduled. The I/O throughput of a VM drops significantly with a consolidation rate of 8 as recommended by VMware [47], based on our evaluation in Section 5.

The I/O inactivity problem becomes even more pronounced when I/O requests are supposed to be processed by fast storage devices (e.g., SSDs). Usually, a vCPU remains active during I/O requests, so it can quickly process them. A similar situation is when a computation task and an I/O task run on the same vCPU. When the I/O task issues a read request and then waits for the request to be satisfied, the computation task is switched on. At this moment, the vCPU is still running; thus, when the read request is satisfied, the vCPU can quickly respond to the event and switch the I/O task back. However, if the time slice of the vCPU is used up by the computation task in one scheduling period, the I/O task cannot proceed until the next period, causing the I/O task to be slowed down significantly.

Worse, the I/O inactivity problem causes the I/O scheduler running in the host OS to work extremely ineffectively. To fully utilize the storage devices, based on the latencies of I/O devices, system designs would carefully control the factors affecting the latencies experienced by I/O workloads (e.g., wake-up latencies and priorities). Thus, I/O workloads running on bare-metal can issue the next request after the previous request is finished. I/O inactive periods make these mechanisms ineffective. Moreover, non-work-conserving I/O schedulers [40] would often hold an I/O request until the next request from the same I/O task comes in (refer to §2.2). By serving the requests from the same task continuously, which have better locality than requests from different tasks, such I/O schedulers can improve I/O throughputs. However, since an I/O workload cannot continue to issue I/O requests after its vCPU becomes inactive, the I/O scheduler in the host OS must switch to serve the requests from other I/O tasks, which greatly reduces locality and I/O throughput, as we will show in our evaluation.

Last but not least, the I/O throughput of a VM can be "capped" by its amount of CPU resource. If the vCPUs in a VM (VM_a) are assigned with smaller proportions of CPU time on each pCPU than the vCPUs on another VM (VM_b), the I/O workloads on VM_a will get less time to issue I/O requests and may only be able to occupy a smaller proportion of the available I/O bandwidth. Since the actual I/O throughputs of the VMs are affected by both I/O scheduling and vCPU scheduling, it is difficult for the I/O scheduler to ensure fairness between the VMs.

All the above problems share the same root cause, I/O inactivity, and existing works mainly try to curb vCPU inactivity but ignore this root cause. Existing works primarily follow two approaches: 1) shortening vCPU time slices (vSlicer [49]); and 2) assigning higher priority to I/O tasks running on active vCPUs (xBalloon [44]). Unfortunately, vCPUs with either approach are still descheduled frequently and cause I/O inactivity.

Since a VM often has active vCPUs, our idea to mitigate I/O inactivity is to try to efficiently migrate I/O tasks to active vCPUs. By evenly redistributing I/O tasks to active vCPUs in a VM, I/O inactivity can be greatly mitigated and I/O tasks can make progress constantly. This maintains both performance and fairness for I/O tasks as they are running on bare-metal. The fairness of I/O bandwidth among VMs on the same host is also maintained.

We implement our idea in vMIGRATER, a user level tool working in each VM. It is transparent as it does not need to modify application, OS in VM, or VMM. vMIGRATER carries simple and efficient mechanisms to predict whether a vCPU will be descheduled and to migrate the I/O tasks on this vCPU to another active vCPU.

vMIGRATER adds only small overhead to applications for two reasons. First, I/O bound tasks use little CPU time, so the I/O tasks migrated by vMIGRATER hardly affect the co-running tasks on the active vCPUs. Second, vMIGRATER migrates more I/O bound tasks to the active vCPUs with more remaining time slices, so all vCPUs' loads in the same VM are well balanced. By reducing I/O inactivity with low overhead, vMIGRATER makes applications run in a fashion similar to what they do on bare-metal, as shown in Figure 1.

vMIGRATER has to address three practical issues. First, it needs to identify I/O tasks. To address this issue, vMIGRATER uses an event-driven model to collect I/O statistics and to detect I/O bound tasks quickly. Second, vMIGRATER needs to determine when an I/O bound task should be migrated. To minimize overhead, vMIGRATER only migrates an I/O bound task when the vCPU running this task is about to be descheduled. vMIGRATER monitors each vCPU's time slice and uses the length of the previous time slice to predict the length of the current time slice. Third, vMIGRATER needs to decide where a task should be migrated to keep it active. Based on the collected time slice and I/O task information, vMIGRATER migrates I/O tasks from to-be-descheduled vCPUs to the active vCPUs based on their remaining time slice and the loads of the tasks.

We implemented vMIGRATER in Linux and evaluated it on KVM [30] with a collection of micro-benchmarks and 7 widely used or studied programs, including small programs (sequential, random and bursty read) from SysBench [7], a distributed file system HDFS [5], a distributed database Hbase [2], a mail server benchmark PostMark [6], a database management system LevelDB [3], and a document-oriented database program MongoDB [35]. Our evaluation shows that:

1. vMIGRATER can effectively improve application throughput. Compared to vanilla KVM, vMIGRATER can improve application throughputs by up to 4.42X. With vMIGRATER, application throughput is 1.84X to 3.64X higher than that with vSlicer or xBalloon.
2. The effectiveness of vMIGRATER increases with consolidation rate. Compared to vanilla KVM, vMIGRATER improves application throughput from 1.72X to 4.42X when the number of consolidated VMs increases from 2 to 8.
3. vMIGRATER can maintain the fairness of the I/O Scheduler in VMM. Compared to vanilla KVM, vMIGRATER reduces unfairness between VMs by 6.22X. When VMs are assigned with the same I/O priority but different CPU time shares, the VMs can still utilize similar I/O bandwidth.

The paper makes the following contributions. First, the paper identifies I/O inactivity as a major factor degrading I/O throughputs in VMs, and quantifies the severity of the problem. Second, it designs vMIGRATER, a simple and practical user-level solution, which greatly improves the throughput of I/O applications in VMs. Third, vMIGRATER is implemented in Linux, and is evaluated extensively to demonstrate its effectiveness.

The remainder of this paper is organized as follows. §2 introduces the background and motivation of vMIGRATER. §3 presents the design principles, architecture, and other design details of vMIGRATER. §4 describes implementation details. §5 presents evaluation results. §6 introduces related work, and §7 concludes the paper.

2 Background and Motivation

This section first introduces vCPU scheduling (§2.1) and I/O request scheduling (§2.2) as the background. Then it explains three performance problems caused by I/O inactivity in virtualized systems (§2.3) to motivate our research.

2.1 vCPU Scheduling

To improve resource utilization in virtualized systems, a pCPU is usually time-shared by multiple vCPUs. A vCPU scheduler is used to periodically deschedule a vCPU and schedule another vCPU. For instance, KVM uses completely fair scheduler (CFS) [13, 44] to schedule vCPUs onto pCPUs. CFS uses virtual runtime (vruntime) to keep track of the CPU time used by each vCPU and to make scheduling decisions. With a red-black tree, it sorts vCPUs based on their vruntime values, and periodically schedules the vCPU with the smallest vruntime value. In this way, CFS distributes time slices to vCPUs in a fair way.

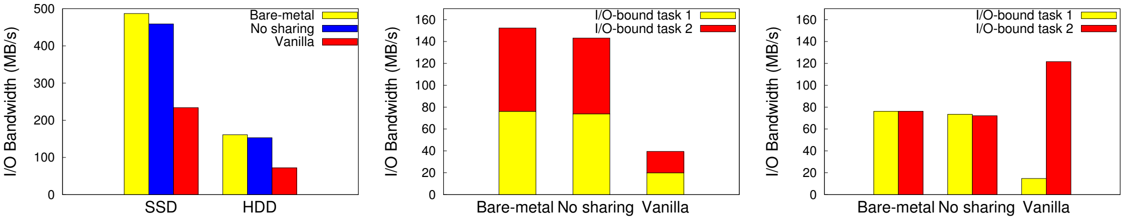
2.2 I/O Request Scheduling

I/O requests are scheduled by the I/O scheduler in the VMM. There are two types of I/O schedulers: work-conserving schedulers [19, 38] and non-work-conserving schedulers [51, 27]. A work-conserving I/O scheduler always keeps the I/O device busy by scheduling pending I/O requests as soon as possible.

Non-work-conserving I/O schedulers, such as anticipatory scheduler (AS) [27] and Completely Fair Queuing (CFQ) [12], are now widely used. A non-work-conserving scheduler waits for a short period after scheduling a request from a task, expecting that other requests from the same task may arrive. Because requests from the same task usually show good locality (i.e., requesting the data at the locations close to each other on the disk), if there are requests from the same task arriving, the scheduler may choose to schedule these requests, even when there are requests from other tasks arriving earlier. It switches to serve the requests from other tasks when the waiting period expires and there are not requests from the same task. Compared to work-conserving I/O schedulers, non-work-conserving schedulers can improve I/O throughput by exploiting locality. The length of waiting periods is selected to balance improved locality and the utilization of I/O devices. To enforce fairness between I/O tasks, an I/O request scheduler controls the distribution of disk time among the tasks.

2.3 Performance Issues Caused by I/O Inactivity

We use experiments to show that serious performance issues will be caused by I/O inactivity. Specifically, we use SysBench [7] to test I/O throughput in three settings. In the *Bare-metal* setting, we run SysBench on the host. In the *No sharing* setting, we run SysBench in a VM; the VM is the only VM in the host. In the *Vanilla* setting, we consolidate 2 VMs on the same host. In the experiments, each VM has 4 vCPUs, and the host has 4 cores. Thus, in the *No-sharing* setting, there is one vCPU on each core, and in the *Vanilla* setting, each core is time-shared by 2 vCPUs. The VMs are configured to have the same I/O bandwidth quota in KVM [30]. In each VM, the CPU workload in SysBench [7] is run as a compute-bound task, and keep



(a) I/O bandwidth of individual VM (b) I/O bandwidth of overall vitualized systems (c) Fairness of I/O Scheduler

Figure 2: ***Three performance issues caused by I/O inactivity*. "Bare-metal" means physical server; "No sharing" means only one VM running on the host; "Vanilla" means two VMs consolidated and managed by vanilla KVM on one host.**

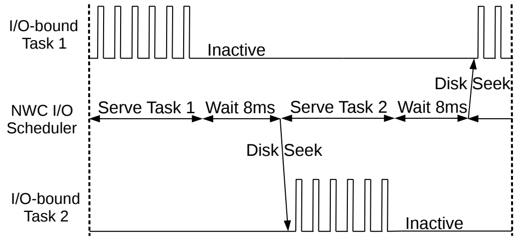


Figure 3: ***I/O inactivity makes Non-Work-Conserving (NWC) I/O scheduler used in VMM ineffective and inefficient because costly disk seeks and waiting time cannot be effectively reduced.***

the vCPUs always busy. Note that we select these workloads and settings mainly to ease the demonstration and analysis of the performance issues. Our evaluation with real workloads and normal settings (§5) show that these performance issues can actually be more severe.

Figure 2 (a) and Figure 2 (b) show that I/O inactivity significantly reduces I/O throughput in two different ways. In the experiment shown in Figure 2 (a), we run only one instance of I/O bound task (i.e., I/O workload of SysBench). Among the three settings, *No sharing* has roughly the same I/O throughput as Bare-metal; but the I/O throughput in the *vanilla* setting is about half of those of the other two settings. This is because the VM running the I/O bound task only obtains 50% of CPU time on each core. Thus, the I/O bound task is only active for 50% of the time, as illustrated in Figure 1.

In the experiment shown in Figure 2 (b), we run two instances of I/O bound task, one in each VM. For brevity, we refer to the I/O bound task in the first VM as I/O bound task 1, and refer to the task in the other VM as I/O bound task 2. The bars in the figure show the throughputs of these tasks, as well as the total I/O bandwidth. As shown in the figure, in the *Vanilla* setting, the total throughput drops by 72.1% compared to bare-metal and no sharing, which is more than 50%.

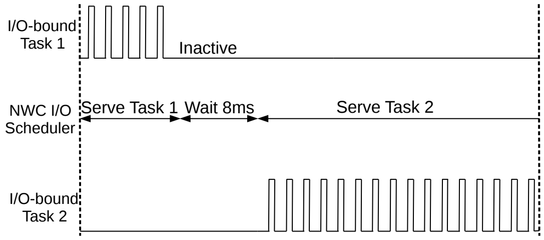


Figure 4: ***I/O inactivity causes unfairness issue.* The I/O task in a VM with more CPU time gets more I/O bandwidth.**

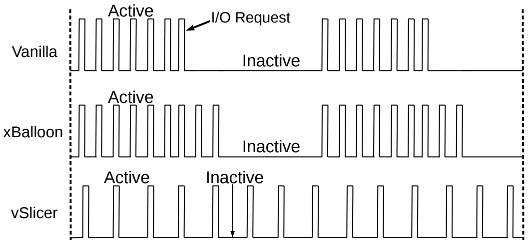


Figure 5: ***The workflow of vanilla, xBalloon and vSlicer.* xBalloon and vSlicer still experience frequent I/O inactivity periods.**

Figure 3 explains the reason. The non-work-conserving I/O scheduler in the VMM serves an I/O bound task in a VM for a short period before the vCPU running the task is descheduled. Then, it waits for 8ms without seeing any requests from I/O bound task 1. Thus, it has to switch and start to serve the I/O-bound task in the other VM (i.e., I/O-bound task 2). The changes between tasks are caused by I/O inactivity. They incur costly disk seeks. The wasted waiting time further reduces I/O throughput.

Figure 2 (c) illustrates the unfairness issue caused by I/O inactivity. It shows that two I/O bound tasks on two VMs with the same I/O priority achieve quite different I/O throughputs because the two VMs are assigned with different CPU time shares. In the experiments, for the *Vanilla* setting, we launch two VMs with the same I/O priority, and run an instance of I/O bound task on each of the VMs. We assign to the VMs with 20% and 80% of CPU time, respectively. For the *Bare-metal* setting and *No sharing* setting, we launch two instances of I/O bound task on the host and the VM, respectively. The two instances of I/O bound task are assigned with the same I/O priority but different CPU time shares (20% and 80%, respectively).

As shown in the figure, the two I/O bound tasks achieve similar I/O throughputs in the *Bare-metal* and *No sharing* settings. However, in the *Vanilla* setting, the I/O bound task in the VM with a larger CPU time share achieves a much higher (5.8x) throughput than that of the I/O bound task in the other VM. Figure 4 explains the cause of this fairness issue. Since VM1 is allocated much less CPU time than VM2, it experiences much longer I/O inactivity periods. As a result, the I/O scheduler serves VM2 for much longer time than VM1.

There are two approaches that may be used to improve I/O throughput. One approach [48, 10, 49] uses smaller time slices (e.g., vSlicer), such that vCPUs are scheduled more frequently, and thus become more responsive to I/O events. As shown in Figure 5 with the curve labeled with vSlicer, this approach reduces the length of each vCPU inactivity period. But I/O inactivity periods become more frequent, and the portion of time in which an I/O task is inactive may not be reduced. Moreover, vSlicer incurs frequent context switches between vCPUs and increases the associated overhead. The other approach [31, 44] lifts the priority of I/O tasks. For example, xBalloon controls how vCPUs consume time slices such that more CPU time can be reserved for the execution of I/O bound tasks on the vCPUs. While this actually lengthens I/O active periods, as shown in Figure 5 with the curve labeled with xBalloon, vCPUs still must be descheduled when they run out of time slices, and I/O inactivity problems are still incurred.

3 vMigrater Design

In this section, we first introduce the design principles and overall architecture of vMigrater. Then, we present the design details of each key component, focusing on how vMigrater monitors the scheduling and descheduling of vCPUs to keep track of their time slices (§3.2), quickly detects I/O-bound tasks (§3.3), and migrates I/O-bound tasks with low overhead (§3.4). Finally, we analyze the performance potential of vMigrater (§3.5).

3.1 Design Principles and Overall Architecture

The design of vMigrater follows three principles:

• Fair: the design of vMigrater must not affect vCPU scheduling (e.g., allocating more CPU time to the vCPUs running I/O tasks) or I/O scheduling in the VMM to avoid any potential unfairness between VMs.

• Non-intrusive: For the wide adoption of vMigrater, the design must be non-intrusive. It should minimize or avoid the modifications to VMM and guest OSs, and should be transparent to applications. Thus, we choose to design vMigrater in the user space of guest OSs. This also helps maintain the original vCPU scheduling and I/O scheduling decisions of the VMM. However, this poses a few challenges, since the migration of I/O bound tasks relies on some key information about vCPU scheduling (e.g., remaining time slice of a vCPU), which is not easy to obtain in the user space.

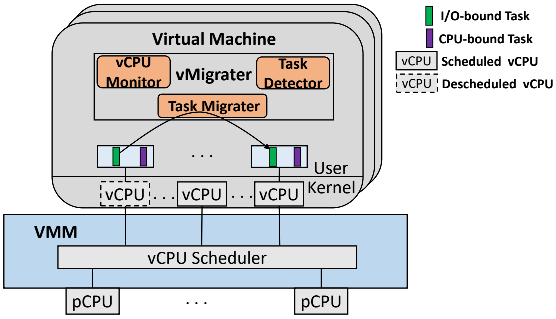


Figure 6: *Overall Architecture of* **vMigrater.**

• Low overhead: vMigrater needs to migrate I/O bound tasks. Frequent migrations may incur high overhead. The design of vMigrater must effectively control the frequency of migrations.

Figure 6 shows the overall architecture of vMigrater and its position in the software stack. vMigrater resides in each VM, and runs at the user level. Following the above principles, three key components are designed as follows.

vCPU Monitor (§**3.2**) monitors the scheduling and descheduling of vCPUs. The objective is to measure time slice lengths for each vCPU and use the lengths to predict whether a vCPU is about to be preempted. The prediction is then used to make decisions on when an I/O bound task should be migrated and where it should be migrated.

Task Detector (§**3.3**) detects I/O activities to quickly determine whether a task is I/O-bound.

Task Migrater (§**3.4)** makes migration decisions and actually migrates I/O-bound tasks. It makes migration decisions based on the vCPU scheduling information from the Task Migrater and I/O activities of the tasks from the Task Detector. Specifically, it tries to migrate an I/O bound task detected in the Task Detector when the vCPU running the task is about to be descheduled. It migrates the task to another vCPU which may not be descheduled in near future.

3.2 vCPU Monitor Design

The vCPU Monitor uses a heartbeat-like mechanism to detect whether a vCPU is running or has been descheduled, with timer events being heartbeats. The idea is that, when a vCPU is descheduled, it cannot process timer events, and the heartbeat pauses. Specifically, vCPU Monitor runs a sleeping thread, namely vCPU Monitor thread, on each vCPU. The sleeping thread is woken up by a timer periodically. When it is woken up, it checks the current clock time, and compare the time with the time it observes last time. A time difference longer than the period for waking up the thread indicates that the vCPU was descheduled earlier, and has just been rescheduled.

This mechanism is as shown in Figure 7. The vCPU Monitor thread can detect that the vCPU is rescheduled at time t_2 and time t_6. The thread keeps track of the timestamps when the vCPU is rescheduled (e.g., t_2 and t_6) and the timestamps immediately before them (e.g., t_1 and t_5). The time slice lengths can be estimated from these timestamps (e.g., $t_5 - t_2$).

Note that, since a vCPU may be scheduled or descheduled while its vCPU Monitor thread is sleeping, the exact time of the vCPU being rescheduled/descheduled cannot be obtained, and thus accurate time slice lengths cannot be measured with this method. Waking up the vCPU Monitor thread more frequently improves the accuracy of estimation; but it increases the overhead at the same time. Considering that typical time slice lengths are tens of miliseconds, vMigrater sets the length of the periods for waking up vCPU Monitor threads to 300 μs to make a trade-off between accuracy and overhead.

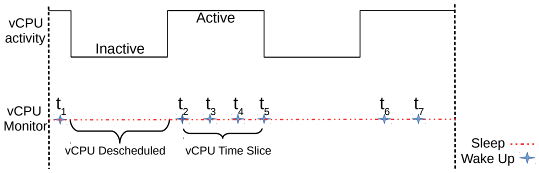


Figure 7: ***vCPU Monitor workflow*****.**

3.3 Detecting I/O bound Tasks

Some applications have bursty I/O operations. Thus, vMigrater needs to quickly respond to workload changes in each application. vMigrater migrates an application when it becomes I/O bound, and stops migration when its I/O phase finishes. However, traditional methods (e.g., Linux top [45] and iotop [26]) for detecting tasks' I/O utilization usually take long time (e.g., seconds). Using these methods may miss the I/O phases in such applications. For instance, the time for an SSD to handle 100MB sequential read is only 100ms. Thus, a much faster method for detecting I/O bound tasks is needed.

vMigrater uses an event-driven method to detect I/O-bound tasks quickly. This method monitors the I/O events triggered by I/O requests, and collects the time spent on processing these I/O events. vMigrater periodically calculates the fraction of time spent on processing I/O events. (The duration of each period in our design is 5 milliseconds.) It determines that a task becomes I/O bound when the fraction exceeds a threshold.

3.4 Migrating I/O bound Tasks

Task Migrater relies the information from vCPU Monitor and Task Detector to make migration decisions. It first needs to decide which I/O tasks should be migrated. To minimize the overhead, Task Migrater only migrates I/O bound tasks when vCPUs running them are to be descheduled shortly. To find these tasks, Task Migrater estimates the remaining time slice for each vCPU[1]. If the remaining time slice is shorter than the length of two periods for waking up vCPU Monitor threads (i.e., 600 μs)[2], Task Migrater determines that the vCPU is about to be descheduled. Task Migrater then checks the tasks scheduled on the vCPU. If there is an I/O bound task reported by Task Detector, Task Migrater migrates the task.

Second, Task Migrater needs to decide which vCPU the I/O bound tasks should be migrated to. A naïve approach is to migrate I/O tasks to the vCPU with the longest remaining time slice. However, this method has two problems if Task Migrater needs to migrate multiple I/O bound tasks: (1) the I/O bound tasks are migrated to the same vCPU and cannot make progress concurrently; (2) the vCPU might be overloaded by accepting all these tasks, and the performance of its existing tasks is degraded.

Task Migrater migrates I/O tasks to vCPUs in a globally balanced way. Specifically, Task Migrater ranks active vCPUs based on the lengths of their remaining time slices, and ranks the I/O bound tasks to be migrated based on their I/O load levels. It migrates the I/O bound tasks with heavier I/O load levels to the vCPUs with longer remaining time slices. This migration mechanism can prevent the above problems because it distributes I/O bound tasks among active vCPUs. At the same time, it helps maintain high I/O throughput because the tasks with the most I/O activities are scheduled on the vCPUs that are least likely to be descheduled shortly.

---

[1] The remaining time slice of a vCPU at a moment (e.g., t_7 in Figure 7) is estimated using the length of time slice assigned to the vCPU before the most recent descheduling of the vCPU (e.g., $t_5 - t_2$) and the CPU time that has already been consumed by the vCPU after the most recent rescheduling of the vCPU (e.g., $t_7 - t_6$).

[2] This is to tolerate the inaccuracy in the estimation of time slices and remaining time slices.

3.5 Performance Analysis

We use Equation (1) to show the performance potential of vMigrater. For simplicity, we assume each VM has at least one active vCPU at any given time. Thus, an I/O application can be kept active with vMigrater, except when it is being migrated.

$$\begin{aligned} Speedup_{vMigrater} &= \frac{T_{ns} \times N}{T_{ns} + N_{migrate} \times C_{avg}} \\ &= \frac{N}{1 + \frac{N_{migrate} \times C_{avg}}{T_{ns}}} \end{aligned} \tag{1}$$

Equation (1) calculates the speedup of an I/O application with vMigrater relative to its execution without vMigrater on a VM. N is the number of vCPUs consolidated on each pCPU (i.e., consolidation rate). T_{ns} is execution time of the I/O application on a VM when its execution is not affected by I/O inactivity problem. This can be achieved by running the application on a vCPU with a dedicated pCPU. It reflects the best performance that an I/O application can achieve on a VM. $N_{migrate}$ is the number of migrations conducted by vMigrater. C_{avg} is the average time cost incurred by each migration.

The numerator of equation (1) is the execution time of an I/O application on a VM without vMigrater. With N vCPUs consolidated on a pCPU, in each period of N time slices, the I/O application can be active only for a period of one time slice. Thus, its execution time is roughly $N \times T_{ns}$. The denominator is the execution time with vMigrater, which is determined by the time spent on application execution and the time spent on migration.

Equation (1) shows that $N_{migrate}$ must be reduced to improve the performance of vMigrater. Suppose vMigrater migrates the I/O application by a minimum number N_{min} of times in an optimal scenario. Thus, $N_{min} = T_{ns}/T_{ts}$, where T_{ts} is the length of a time slice allocated to a vCPU. In this optimal scenario, the I/O application is moved to a vCPU when the vCPU is just rescheduled; it stays there until the timeslice of the vCPU is used up; it is then moved to another vCPU which is newly rescheduled.

Replacing T_{ns} with $T_{ts} \times N_{min}$ in equation (1), we get:

$$Speedup_{vMigrater} = \frac{N}{1 + \frac{N_{migrate} \times C_{avg}}{N_{min} \times T_{ts}}} \tag{2}$$

Equation 2 shows that the speedup is determined by N and $\frac{N_{migrate} \times C_{avg}}{N_{min} \times T_{ts}}$; N, C_{arg}, and T_{ts} are constants for an application. We denote $\frac{N_{migrate}}{N_{min}}$ as $P_{vMigrater}$, which has a value greater than 1. The speedup is mainly determined by $P_{vMigrater}$. When $P_{vMigrater}$ approaches to 1, the speedup approaches to N. Our experiments show that the speedup with vMigrater matches the speedup calculated by Equation 1.

4 Implementation Details

We have implemented vMigrater on Linux. The implementation of vCPU Monitor relies on a reliable and accurate clock source to generate timer events. The traditional system time clock cannot satisfy this need when vCPUs time-share a pCPU [44]. Instead, we use the clock source CLOCK_MONOTONIC [18], which is more reliable and can provide more accurate time measurement. The implementation of Task Detector leverages BCC [11, 24] to monitor I/O requests. BCC is a toolkit supported by Linux kernel for creating efficient kernel tracing and manipulation programs.

The implementation of Task Migrater uses two mechanisms, PUSH and PULL, to migrate tasks. A PUSH operation is conducted by the source vCPU of a task to move the task to the destination vCPU, while a PULL operation is initiated by the destination vCPU to move a task to it from the source vCPU. Usually PUSH operations are used. PULL operations are only used when source vCPUs are descheduled and cannot conduct PUSH operations. vMigrater's source codes are available on `github.com/hku-systems/vMigrater`.

5 Evaluation

Our evaluation is done on a DELL™ PowerEdge™ R430 server with 64GB of DRAM, one 2.60GHz Intel® Xeon® E5-2690 processor with 12 cores, a 1TB HDD, and a 1TB SSD. All VMs (unless specified) have 12 vCPUs and 4GB memory. The VMM is KVM [30] in Ubuntu 16.04. The guest OS in each VM is also Ubuntu 16.04. The length of a vCPU time slice is 11ms, and the wait time of the CFQ I/O scheduler in VMM is set to 8ms, as recommended by Red Hat [41, 42, 40].

We evaluate vMigrater using a collection of micro-benchmarks and 7 widely used applications. Micro- benchmarks include *SysBench [7] sequential read*, *SysBench random read*, and *bursty read* implemented by us. As summarized in Table 1, applications include *HDFS* [5], *LevelDB* [3], *MediaTomb* [9], *HBase* [2], *PostMark* [6], *Nginx* [37], and *MongoDB* [35]. To be close to real-world deployments, `PostMark` is run with `ClamAV` (antivirus program) [17] to generate the workload of a complete mail server with antivirus support; *LevelDB* and *MongoDB* are deployed as the back-end storage of a Spark [52] system.

Application	**Workload**
`HDFS`	Sequentially read 16GB with `HDFS` `TestDFSIO` [25].
`LevelDB`	Randomly scan table with db_bench [4].
`MediaTomb`	Concurrent requests on transcoding a 1.1GB video.
`HBase`	Randomly read 1GB with `HBase` `PerfEval` [25].
`PostMark`	Concurrent requests on a mail server.
`Nginx`	Concurrent requests on watermarking images [1].
`MongoDB`	Sequentially scan records with YCSB [8].

Table 1: ***7 applications and workloads.***

Most of the experiments are conducted with the SSD. Only the experiments in §5.4 (fairness of I/O scheduler) use the HDD, because they need a non-work-conserving I/O scheduler (e.g., CFQ) and CFQ is used in Linux to schedule HDD requests.

We compare vMigrater with two related solutions: xBalloon [44] and vSlicer [49]. Because they do not have open-source implementations, we implemented them based on the description in the corresponding papers.

Our evaluation aims to answer the following questions:

§5.1: Is vMigrater easy to use?

§5.2: How much performance improvement can be achieved with vMigrater, compared with vanilla KVM and two related solutions? What is the overhead incurred by vMigrater?

§5.3: What is vMigrater's performance when the workload in a VM varies over time?

§5.4: Can vMigrater help the I/O scheduler in the VMM to achieve fairness between VMs?

5.1 Ease of Use

With vMigrater, all 7 real applications we evaluated could run smoothly without any modification. When we evaluate these applications, vMigrater runs at the user-level of the guest OS. There is no need to change any parts of the VM or the VMM.

5.2 Performance Improvements

We first demonstrate that vMigrater can greatly improve the throughput of I/O intensive applications in each VM. For this purpose, we vary the number of VMs hosted on the server from 1 to 8. The co-located VMs run the same workloads. Specifically, we run the workloads with the micro-benchmarks and the real applications in the VMs. For each workload, we run one instance of the workload in each VM. When we run *HDFS*, *LevelDB*, *HBase*, and *MongoDB*, we also run the `TeraSort` benchmark [25] in each VM, which is compute-bound. We measure the throughputs of the benchmarks and real applications. When only one VM is hosted on the server, the I/O inactivity problem does not happen; the benchmarks and applications achieve the highest performance. We refer to this setting as **No sharing**, and use the performance under this setting as reference performance. We normalize the performance under other settings (i.e., 2/4/8 VMs consolidated on the server) against the reference performance, and show the normalized performance. Thus, the normalized performance of 1 is the best performance that can be achieved. The closer the normalized performance is to 1, the better the performance is.

Figure 8 shows the normalized throughputs for micro-benchmarks when the number of consolidated VMs is varied from 2 to 8. With vMigrater, the benchmarks consistently achieve better performance than they do on vanilla KVM. At the same time, the performance advantage with vMigrater becomes more prominent when more VMs are consolidated. On average, with vMigrater, the throughputs of these benchmarks are improved by 97%, 225%, and 431% than those on vanilla KVM for the settings with 2 VMs, 4 VMs, and 8VMs, respectively.

Similar performance improvements are also observed with real applications, as shown in Figure 9. On average, with vMigrater, the throughputs of these applications are 72%, 192%, and 342% higher than those on vanilla KVM for the settings with 2 VMs, 4 VMs, and 8VMs, respectively.

Compared to vSlicer and xBalloon, the applications can also achieve better performance with vMigrater. As shown in Figure 9, On average, with vMigrater, the throughputs of these applications are 88.41%, 74.86%, and 121.22% higher than those with vSlicer for the settings with 2 VMs, 4 VMs, and 8VMs, respectively; and the throughputs are 3.29%, 83.78%, and 175.37% higher than those with xBalloon under these three settings.

vMigrater can significantly improve the throughput of I/O applications when all the consolidated VMs are equipped with vMigrater. However, since vMigrater is implemented at the user space, it is possible that not all the VMs have vMigrater deployed. We wonder whether vMigrater can still effectively improve performance in this scenario. To answer this question, we run *HDFS* or *LevelDB* in one of the collocated VMs and enables vMigrater in this VM; in other VM(s), we run the IS benchmark in NPB benchmark suite [36], and disable vMigrater in the VM(s). Figure 10 shows that the effectiveness of vMigrater is not affected. On average, with vMigrater, the throughputs of these applications are improved by 62.72%, 176.92%, and 218.75% than those on vanilla KVM for the settings with 2 VMs, 4 VMs, and 8VMs, respectively.

To understand how the performance improvements are achieved, we profile the executions of the

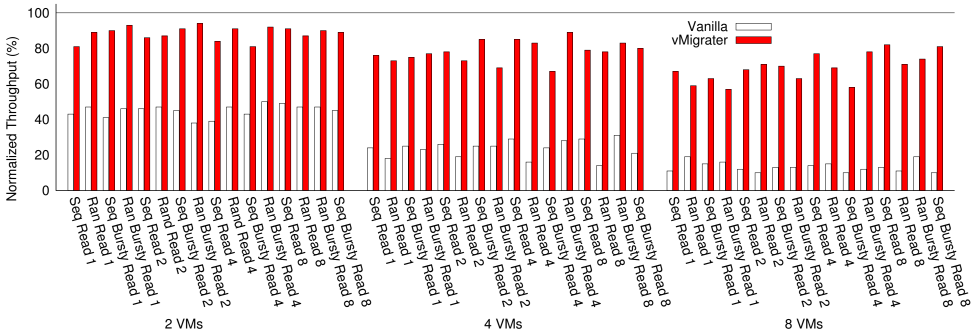


Figure 8: ***Normalized throughputs of micro-benchmarks***

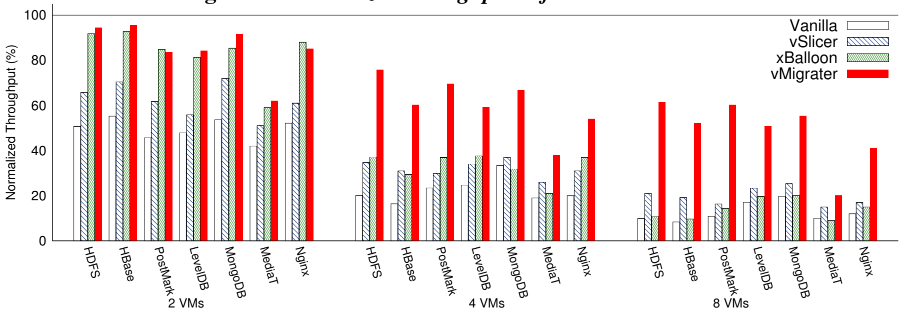


Figure 9: ***Normalized throughput of real applications***

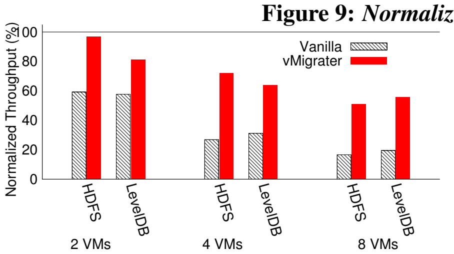


Figure 10: ***Normalized throughputs of `HDFS` and `LevelDB` when* VMIGRATER *is enabled in one of the consolidated VMs.***

real applications. We collect the number of migrations and the time during which I/O bound tasks "run" on descheduled vCPUs (i.e., I/O inactivity time). We show the data in Table 2 and Table 3.

VMIGRATER greatly improves application performance by first dramatically reducing I/O inactivity time. As shown in Table 2, on average, VMIGRATER reduces I/O inactivity time by 860.27%, 657.87%, 562.92%, respectively, relative to vanilla KVM, vSlicer, and xBalloon.

When I/O inactivity time has been dramatically reduced, as we have analyzed in Section 3.5, VMIGRATER maintains high throughputs by minimizing the time spent on migrating tasks, which is determined by the number of migrations and the time to finish each migration. As shown in Table 3, for most applications, the $P_{vMigrater}$ values are very close to 1. This confirms that the migration mechanisms in VMIGRATER are well designed. On one hand, they have effectively migrated I/O bound tasks to keep them active and minimize I/O inactivity. On the other hand, they only migrate the tasks for close-to-minimal times, so as to keep the time spent on migration low. We notice that the $P_{vMigrater}$ value is the highest (1.34) for `MediaTomb` among these applications, and its Speedup is the lowest (1.41). This confirms the performance analysis in Section 3.5.

We also notice that the I/O throughputs that applications can achieve reduce slightly when the consolidation rate increases. This is caused by the special design with the vCPU scheduler in KVM (i.e., CFS in Linux), which allocate smaller time slices with higher consolidation rates. This reduces the opportunity to migrate I/O bound tasks. This problem can be mitigated by waking up Task Detector threads more frequently.

Application	Vanilla	vSlicer	xBalloon	vMigrater	Ratio
`HDFS`	121.82s	92.91s	75.27s	6.62s	18.39
`LevelDB`	129.45s	101.55s	79.84s	17.86s	7.25
`HBase`	98.13s	69.37s	75.71s	18.93	5.19
`MongoDB`	39.49s	30.34s	40.57s	3.49s	11.31
`PostMark`	225.32s	168.01s	113.01s	12.92s	17.44
`MediaTomb`	108.61s	89.46s	116.96s	34.95s	3.11
`Nginx`	59.15s	61.72s	42.37s	8.03s	7.37

Table 2: ***I/O inactivity time (seconds) of 7 applications.*** **Four VMs are used. The last column is the ratio between the I/O inactive time with vanilla KVM and that with VMIGRATER.**

Application	$N_{migrate}$	N_{min}	$P_{vMigrater}$	Speedup
`HDFS`	3363	3181	1.05	1.86
`LevelDB`	2154	2003	1.07	1.75
`HBase`	3454	3181	1.08	1.76
`MongoDB`	1545	1363	1.13	1.70
`PostMark`	5181	4818	1.07	1.82
`MediaTomb`	2454	1818	1.34	1.41
`Nginx`	4181	4090	1.02	1.73

Table 3: **VMIGRATER** ***only migrates I/O bound tasks for close-to-minimal times.*** **Two VMs are used.**

Improved I/O throughputs can translate to lower response times. Figure 11 shows normalized re-

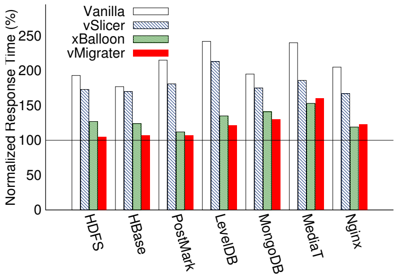


Figure 11: ***Normalized response times of*** `HDFS`***,*** `HBase`***,*** `PostMark`***,*** `LevelDB`***,*** `MongoDB`***,*** `MediaTomb`***, and*** `Nginx`***.*** **Two VMs are used.**

sponse time of the real applications with vanilla KVM, vSlicer, xBalloon, and vMIGRATER. With vanilla KVM, the response time is almost doubled when 2 VMs time-share the server, relative to that under the "No sharing" setting. With vSlicer, the response times of the applications can be slightly lowered by 11.56% relative to those with vanilla KVM. xBalloon and vMIGRATER can significantly reduce response times (61.97% and 66.67% on average for the applications, relative to those with vanilla KVM). With xBalloon, response times are low mainly because the applications are given higher priorities on active vCPUs, and thus can proceed more quickly when their vCPUs are active. With vMIGRATER, response times are low mainly because, with minimized I/O inactivity, the applications can proceed constantly. As a result, the responses times with vMIGRATER are almost the same to those under the "No sharing" setting.

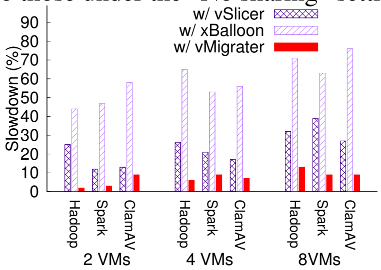
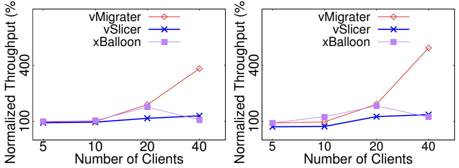


Figure 12: ***Slowdowns of compute-bound applications with vSlicer, xBalloon, and*** **VMIGRATER*****, relative to their executions with vanilla KVM.***

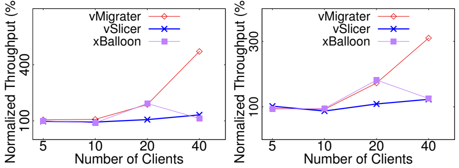
An important requirement on the solutions for improving the performance of I/O-bound applications on VMs is minimal impact on other applications. We have examined the impact on compute-bound applications by comparing the performance of *Hadoop*, *Spark*, and `ClamAV` with each of vSlicer, xBalloon, and vMIGRATER against that with the vanilla KVM. *Hadoop* is a Hadoop TeraSort workload; *Spark* is a Spark WordCount workload; `ClamAV` performs virus scanning. We vary the number of VMs from 2 to 8. In each VM, one of these workload is co-run with an I/O bound workload (`PostMark` with `ClamAV`, `HDFS` with *Hadoop*, and `LevelDB` with *Spark*). Each VM has 12 vCPUs.

Figure 12 shows that the slowdowns with vMIGRATER are smaller than those with vSlicer and xBalloon consistently across all the scenarios. On average, the slowdowns are 7.44%, 23.56% and 59.22% with vMIGRATER, vSlicer and xBalloon, respectively. The slowdowns are large with vSlicer and xBalloon because xBalloon prioritizes I/O-bound tasks and delays compute-bound tasks, and vSlicer increases costly vCPU switches.

5.3 Robustness to Varying Workloads



(a) HDFS (b) Hbase



(c) PostMark (d) MongoDB

Figure 13: ***Throughput improvement of*** `HDFS`***,*** `HBase`***,*** `PostMark`***, and*** `MongoDB` ***with vSlicer, xBalloon, and*** **VMIGRATER*****, in VMs with increasing workload, relative to the executions with vanilla KVM.***

We show with experiments that vMIGRATER is robust and can tolerate workload variation in VMs. In the experiments, We run two 2-vCPU VMs on two pCPUs. In each VM, we run an I/O-bound workload (`HDFS`, `HBase`, `PostMark`, or `MongoDB`). We also runs a set of threads on a separate computer, each of which emulates a *client* submitting compute-bound tasks to the VMs in a back-to-back manner. To serve multiple clients concurrently, the CPU usage of the compute-bound tasks submitted by a client is throttled below 20% in each VM. The compute-bound tasks are `TeraSort` in the VMs with `HDFS`, `HBase`, and `MongoDB`, and `ClamAV` in the VMs with `PostMark`. We increase the number of clients from 5 to 40, and measure the throughputs of the I/O-bound workload.

Figure 13 shows the throughput improvement of the I/O-bound applications with vSlicer, xBalloon, and vMIGRATER, relative to the executions with vanilla KVM. Before the number of clients reaches 10, vMIGRATER can barely improve I/O throughputs; but it does not degrade performance. This is because the VMs are not fully loaded in these cases, and their vCPUs cannot completely consume their time-slices. Thus, the I/O inactivity problem rarely happens. When the number of clients keeps growing, the I/O inactivity problem becomes increasingly serious; there is more potential for vMIGRATER to improve I/O throughputs; and vMIGRATER becomes more effective. When the number of clients reaches 40, the throughputs with the vanilla drop significantly; with vMIGRATER, the applications can still maintain their throughputs as high as those with a few clients.

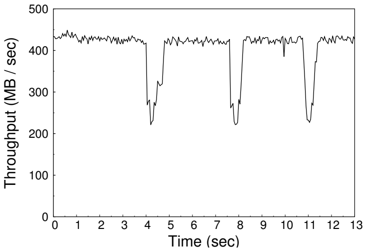
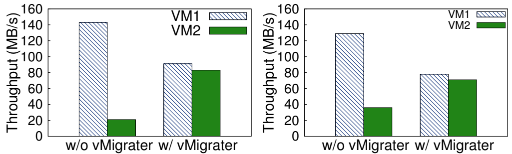


Figure 14: ***Throughput changes of HDFS when the number of clients increases from 8 to 36.***

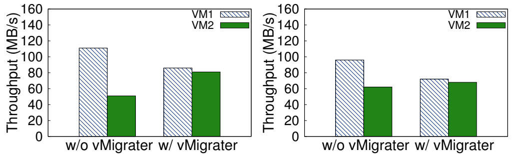
To demonstrate that vMIGRATER can quickly respond to workload changes, Figure 14 shows how the throughput of *HDFS* changes when we increase the number of clients from 8 to 36 in 11 seconds. The clients are added at different rates (4 at the time of about 4 seconds, 8 at about 8 seconds, and 16 at about 11 seconds) to generate different levels of workload change intensity. As shown in the figure, at the moments when the workload changes in the VM, because vMIGRATER cannot correctly estimate the length of time slices, the throughput of *HDFS* drops temporarily. However, vMIGRATER can quickly adapt and the throughput picks up in about a half second.

5.4 Maintaining Fairness



(a) VM1: 90%; VM2: 10% CPU

(b) VM1: 80%; VM2: 20% CPU



(c) VM1: 70%; VM2: 30% CPU

(d) VM1: 60%; VM2: 40% CPU

Figure 15: ***Throughputs of HDFS in the two VMs with equal I/O bandwidth shares but different CPU time shares.***

To demonstrate that vMIGRATER can help maintaining I/O fairness, we consolidate two 12-vCPU VMs and make them share 12 pCPUs. Each VM runs an instance of `HDFS` (I/O-bound) and an instance of `TeraSort` (compute-bound) concurrently. We adjust the system settings controlling the I/O bandwidth shares and CPU time shares of the VMs, so that equal shares of I/O bandwidth but different shares of CPU time are allocated to the VMs. Figure 15 shows the I/O throughputs achieved by the two instances of `HDFS` in the two VMs, when the CPU time share of one VM is varied from 90% to 60% and the other VM uses the remaining share of CPU time. Without vMIGRATER, there is a strong correlation between I/O throughputs and CPU time shares, despite that the two instances are expected to achieve similar I/O throughputs. With vMIGRATER, the I/O throughputs are largely independent of the CPU time shares of the two VMs. The I/O throughput in the VM with a larger share of CPU time is slightly higher, because the I/O-bound tasks in the VM are migrated less frequently.

6 Related Work

Shortening time slices. Many efforts have focused on shortening the time slices of vCPUs [10, 49, 48] in order to process I/O events more frequently. There are two issues with these solutions: (1) I/O inactivity periods still exist and degrade I/O performance; (2) frequent context switches between vCPUs are caused and degrade system performance [21, 46, 32]. These solutions require intensive modifications to both the VMM and guest OS kernel.

Dedicating CPUs. Dedicating CPUs [43, 14] aims to solve the problems caused by vCPUs contending pCPU resource by reducing the number of vCPUs sharing a pCPU or dedicating a pCPU to each vCPU. However, this may lower the utilization of hardware resource and overall system throughput.

Task-aware priority boosting. There are designs [21, 31, 15, 44, 39, 20, 29, 50, 23, 34, 22, 33, 16, 28] to prioritize latency-sensitive tasks to improve their performance. For example, task-aware VM scheduling [31] prioritizes I/O bound VMs; xBalloon preserves the priority of I/O tasks by preserving CPU resource for I/O tasks [44]. However, the vCPUs are still descheduled with these designs; so the I/O inactivity periods still exist.

7 Conclusion and Future Work

This paper identifies I/O inactivity problem in VMs which has not been adequately studied before. It presents vMIGRATER, a simple, fast and transparent system that can greatly mitigate I/O inactivity. vMIGRATER mainly aims to mitigate the performance degradation caused by disk (HDD or SSD) I/O inactivity periods in VMs. As future work, we will extend it for network I/O.

Acknowledgments

We thank anonymous reviewers for their helpful comments. This work is funded in part by the US National Science Foundation under grants CCF 1617749 and CNS 1409523, research grants from Huawei Innovation Research Program 2017, HK RGC ECS (27200916), HK RGC GRF (17207117), HK CRF grant (C7036-15G), and a Croucher innovation award.

References

[1] Adding watermarks to images using alpha channels. `http://php.net/manual/en/image.examples-watermark.php`.

[2] HBase. `https://hbase.apache.org/`.

[3] LevelDB. `https://github.com/google/leveldb`.

[4] LevelDB Benchmarks. `http://www.lmdb.tech/bench/microbench/benchmark.html`.

[5] The Hadoop Distributed File System. `http://hadoop.apache.org/hdfs/`.

[6] The PostMark Benchmark. `http://www.filesystems.org/docs/auto-pilot/Postmark.html`.

[7] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`, 2004.

[8] Yahoo! Cloud Serving Benchmark. `https://github.com/brianfrankcooper/YCSB`, 2004.

[9] MediaTomb - Free UPnP MediaServer. `http://mediatomb.cc/`, 2014.

[10] J. Ahn, C. H. Park, and J. Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 394–405. IEEE Computer Society, 2014.

[11] `https://iovisor.github.io/bcc/`.

[12] `https://en.wikipedia.org/wiki/CFQ`.

[13] `https://en.wikipedia.org/wiki/Completely_Fair_Scheduler`.

[14] L. Cheng, J. Rao, and F. Lau. vscale: automatic and efficient processor scaling for smp virtual machines. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 2. ACM, 2016.

[15] L. Cheng and C.-L. Wang. vbalance: using interrupt load balance to improve i/o performance for smp virtual machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 2. ACM, 2012.

[16] R. C. Chiang and H. H. Huang. Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2011.

[17] `http://www.clamav.net/`.

[18] `http://man7.org/linux/man-pages/man2/timer_create.2.html`.

[19] `https://en.wikipedia.org/wiki/Deadline_scheduler`.

[20] X. Ding, P. B. Gibbons, and M. A. Kozuch. A hidden cost of virtualization when scaling multicore applications. In *HotCloud*, 2013.

[21] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, pages 73–84, 2014.

[22] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu. Opportunistic flooding to improve tcp transmit performance in virtualized clouds. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 24. ACM, 2011.

[23] S. Gamage, C. Xu, R. R. Kompella, and D. Xu. vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[24] B. Gregg. Performance superpowers with enhanced BPF. Santa Clara, CA, 2017. USENIX Association.

[25] Hadoop. `http://hadoop.apache.org/core/`.

[26] `https://linux.die.net/man/1/iotop`.

[27] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 117–130. ACM, 2001.

[28] W. J. Jianchen Shan and X. Ding. Rethinking the scalability of multicore applications on big virtual machines. In *IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2017.

[29] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[30] `http://www.linux-kvm.org/`.

[31] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110. ACM, 2009.

[32] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 2007.

[33] H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu. vfair: Latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 125–138. ACM, 2015.

[34] H. Lu, C. Xu, C. Cheng, R. Kompella, and D. Xu. vhaul: Towards optimal scheduling of live multi-vm migration for multi-tier applications. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 453–460. IEEE, 2015.

[35] Mongodb. `http://www.mongodb.org`, 2012.

[36] `http://www.nas.nasa.gov/publications/npb.html`.

[37] Nginx web server. `https://nginx.org/`, 2012.

[38] `https://en.wikipedia.org/wiki/Noop_scheduler`.

[39] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[40] Red Hat. What is the suggested I/O scheduler to improve disk performance (2017). `https://access.redhat.com/solutions/5427`.

[41] RedHat. `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/6.0_technical_notes/deployment`.

[42] RedHat. `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/index`.

[43] X. Song, J. Shi, H. Chen, and B. Zang. Schedule processes, not vcpus. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 1. ACM, 2013.

[44] K. Suo, Y. Zhao, J. Rao, L. Cheng, X. Zhou, and F. Lau. Preserving i/o prioritization in virtualized oses. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 269–281. ACM, 2017.

[45] `http://man7.org/linux/man-pages/man1/top.1.html`.

[46] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Proceedings of the 2007 workshop on Experimental computer science*, page 4. ACM, 2007.

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

[48] C. Xu, S. Gamage, H. Lu, R. R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *USENIX Annual Technical Conference*, pages 243–254, 2013.

[49] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM, 2012.

[50] C. Xu, B. Saltaformaggio, S. Gamage, R. R. Kompella, and D. Xu. vread: Efficient data access for hadoop in virtualized clouds. In *Proceedings of the 16th Annual Middleware Conference*, pages 125–136. ACM, 2015.

[51] Y. Xu and S. Jiang. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics. In *FAST*, pages 119–132, 2011.

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.