ABSTRACT

IMPROVING MULTI-THREADED QOS IN CLOUDS

by
Weiwei Jia

Multi-threading and resource sharing are pervasive and critical in clouds and data-centers. In order to ease management, save energy and improve resource utilization, multi-threaded applications from different tenants are often encapsulated in virtual machines (VMs) and consolidated on to the same servers. Unfortunately, despite much effort, it is still extremely challenging to maintain high quality of service (QoS) for multi-threaded applications of different tenants in clouds, and these applications often suffer severe performance degradation, poor scalability, unfair resource allocation, and so on.

The dissertation identifies the causes of the QoS problems and improves the QoS of multi-threaded execution with three approaches. First, the dissertation identifies that the I/O performance of an application can be significantly affected by its computation on VMs. Particularly, the I/O inactivity problem is caused when the computation workload has consumed the CPU time allocated to virtual CPUs (vCPUs), preventing the I/O workload on these vCPUs from producing I/O requests. This problem can greatly degrade I/O performance and cause fairness issues in I/O scheduling. Second, on modern simultaneous multi-threading (SMT) processors, existing CPU schedulers are ineffective to schedule I/O workloads for high I/O performance and efficiency. Third, due to frequent synchronization and communication, multi-threaded workloads are more vulnerable to the contention for CPU time in clouds. As a result, these workloads suffer severe performance degradation and interference.

The dissertation presents three systems, vMIGRATER, vSMT-IO, and JUPITER, with each addressing a distinct research problem. Extensive evaluations on diverse multi-threaded applications, including DBMS, web servers, AI workloads, Hadoop jobs, and so on, show that these systems can significantly improve the QoS of multi-threaded applications in clouds.

IMPROVING MULTI-THREADED QOS IN CLOUDS

by
Weiwei Jia

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2021

APPROVAL PAGE

IMPROVING MULTI-THREADED QOS IN CLOUDS

Weiwei Jia

Xiaoning Ding, PhD, Dissertation Advisor | Date
Associate Professor, Computer Science, NJIT

Cristian Borcea, PhD, Committee Member | Date
Professor, Computer Science, NJIT

Jason Tsong-Li Wang, PhD, Committee Member | Date
Professor, Computer Science, NJIT

Jing Li, PhD, Committee Member | Date
Assistant Professor, Computer Science, NJIT

Qing Liu, PhD, Committee Member | Date
Assistant Professor, Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author:	Weiwei Jia
Degree:	Doctor of Philosophy
Date:	May 2021

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, USA, 2021
- Master of Science in Computer Science,
  Northwestern Polytechnical University, ShaanXi, China, 2016
- Bachelor of Science in Software Engineering,
  Xi'An University of Posts and Telecommunications, ShaanXi, China, 2013

Major: Computer Science

Presentations and Publications:

W. Jia, J. Shan, J. Li, X. Ding, "Effective and QoS-aware Coscheduling for Multi-threaded Workloads in Multi-Tenant Clouds". *(under submission)*

W. Jia, J. Shan, T. Li, X. Shang, H. Cui, X. Ding, "vSMT-IO: Improving I/O Performance and Efficiency on SMT Processors in Virtualized Clouds", *in 31th USENIX Annual Technical Conference (USENIX ATC 2020)*, 2020.

W. Jia, C. Wang, X Chen, J. Shan, H. Cui, X. Ding, L. Cheng, F. Lau, Y. Wang, "Effectively Mitigating I/O Inactivity in vCPU Scheduling", *in 29th USENIX Annual Technical Conference (USENIX ATC 2018)*, 2018.

C. Wang, X. Chen, W. Jia, B. Li, H. Qiu, S. Zhao, H. Cui, "PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance", *in 15th USENIX Symposium on Networked Systems Design and Implementation (USENIX NSDI 2018)*, 2018.

J. Shan, W. Jia, X. Ding, “Rethinking the scalability of multicore applications on big virtual machines”, *in IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS 2017)*, 2017.

To my family.

ACKNOWLEDGMENT

First and foremost, I extend special thanks to my advisor, Prof. Xiaoning Ding, for his support and direction in all aspects of my PhD career. When I try to describe what I learnt from him and my gratitude, I find it's hard to describe with simple words. This is similar to someone wants you to describe your gratitude to your parents and what you learnt from them. In the rest of my life, I believe we will keep discussing research problems, chatting, and many more, as usual.

I also extend thanks to the PhD committee members, Prof. Cristian Borcea, Prof. Jason Tsong-Li Wang, Prof. Jing Li and Prof. Qing Liu. Special thanks to Prof. Jing Li for her direction and guidance on Jupiter project, and her suggestions on how to do a good research. I also enjoy chatting with Prof. Jason Tsong-Li Wang in his office; thanks for your time.

I would also like to take this opportunity to thank the Department of Computer Science at NJIT for the teaching assistant scholarship support. The department also provides great resources and environments. I feel very lucky to be a Ph.D. student in the department.

I also sincerely thank Prof. Heming Cui, Prof. Yuexuan Wang, Prof. Francis C. M. Lau, Dr. Jianchen Shan, Dr. Luwei Cheng, Cheng Wang, Xusheng Chen, Jianyu Jiang, Shixiong Zhao, Tze On Li, Xiangjun Peng, Yuangang Wang for their valuable suggestions, advice and collaboration during my PhD study.

At last, I am grateful to my family. My parents and my wife give me endless love and support whenever I need them. My wife also leaves me much time for my research. Without their support, this dissertation may not exist.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter Page

TABLE OF CONTENTS
(Continued)

Chapter Page

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure Page

LIST OF FIGURES (Continued)

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

The essence of cloud computing is twofold. First, sharing resources among tenants can reduce cost. Second, scaling applications can achieve high performance. Thus, time-sharing resources and multi-threading are pervasive and critical in clouds and data-centers. As has been widely observed, applications in virtual machines (VMs) use increasingly more threads, and VMs sharing the same physical server use increasingly more virtual CPUs (vCPUs), in order to exploit parallelism on multi-core processors. For instance, in Amazon EC2, a physical server can be shared by multiple virtual instances, and a virtual instance can have as many as 128 vCPUs [1; 2].

Unfortunately, despite much effort from both academia and industry [3; 4; 5; 6; 7; 8; 9; 10], the quality of service (QoS), as a key metric of cloud computing service, is still notoriously bad for multi-threaded applications. The applications usually suffer significant performance degradation, poor scalability, unfair resource allocation, and so on. Our research reveals that the bad QoS is caused by three major reasons.

First, the I/O performance of an application can be significantly affected by its computation on virtualized platforms, such as clouds and data-centers. Since CPU cores are time-shared by vCPUs, vCPUs are scheduled and descheduled by the hypervisor of VMs periodically. In each 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. For instance, the utilization of I/O resource in the guest OS is low during I/O inactivity periods. Worse, the I/O scheduler in the host OS suffers from low performance because the I/O scheduler assumes that I/O tasks make I/O requests constantly. Existing works typically adjust time slices of vCPUs running I/O tasks, but these vCPUs can still be inactive frequently, and fairness issues among VMs within a host can easily occur. We present vMIGRATER (Chapter 2) to solve the I/O inactivity problem.

Second, CPU resources are not efficiently utilized on modern processors supporting simultaneous multi-threading (SMT, e.g., hyper-threading on X86 processors). Because existing CPU scheduler designs fail to fully consider the hardware features of SMT processors, three problems may be caused when scheduling vCPUs on SMT processors: 1) idle/spinning vCPUs are handled inefficiently with high overhead, reducing system and per-VM throughput; 2) the descheduling of idle/spinning vCPUs are unnecessarily frequent, delaying the operations on these vCPUs and causing high latency; 3) the even distribution of workload to the hardware threads on the same core cannot effectively reduce the resource contention between the vCPUs running on different hardware threads, further reducing throughput. We present vSMT-IO (Chapter 3) to make vCPU scheduler SMT aware.

Third, due to frequent synchronization and communication, multi-threaded workloads are more vulnerable to the contention for CPU time in time-shared clouds. As a result, these workloads suffer severe performance degradation and interference. Though co-scheduling [11] is an effective approach for improving multi-threaded performance in time-shared environments, its adverse impacts, such as high context switch overhead, CPU fragmentation, and priority inversion, are also notoriously difficult to mitigate without sacrificing its effectiveness on improving multi-threaded performance. Thus, achieving desirable performance for multi-threaded applications in time-shared clouds remains an open problem [5; 12; 13]. We present a framework, JUPITER (Chapter 4), to improve multi-threaded QoS while minimizing the adverse impacts caused by co-scheduling in clouds.

The research contributions are summarized in the following sections.

1.2 Contributions of Dissertation

1.2.1 VMIGRATER: Effectively Mitigating I/O Inactivity in vCPU Scheduling

The dissertation presents VMIGRATER, which runs in the user level of each VM. The idea is that each VM often has active vCPUs, and we can efficiently migrate I/O tasks to these vCPUs, greatly mitigating the I/O inactivity periods and maintaining fairness. It introduces 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 be up to 4.42X faster than the default Linux KVM. VMIGRATER is also 1.84X to 3.64X faster than two related systems.

1.2.2 VSMT-IO: Making Virtual CPU Scheduler SMT-Aware for Lower Latency and Higher Throughput

The dissertation focuses on an under-studied yet fundamental issue on Simultaneous Multi-Threading (SMT) processors — how to schedule I/O workloads, so as to improve I/O performance and efficiency. The paper shows that existing techniques used by CPU schedulers to improve I/O performance are inefficient on SMT processors, because they incur excessive context switches and spinning when workloads are waiting for I/O events. Such inefficiency makes it difficult to achieve high CPU throughput and high I/O throughput, which are required by typical workloads in clouds with both intensive I/O operations and heavy computation.

The dissertation proposes to use *context retention* as a key technique to improve I/O performance and efficiency on SMT processors. Context retention uses a hardware thread to hold the context of an I/O workload waiting for I/O events, such that overhead of context switches and spinning can be eliminated, and the workload can quickly respond to I/O events. Targeting virtualized clouds and x86 systems, the paper identifies the technical issues in implementing context retention in real systems, and explores effective techniques to address these issues, including long term context retention and retention-aware symbiotic scheduling.

The dissertation designs vSMT-IO to implement the idea and the techniques. Extensive evaluation based on the prototype implementation in KVM and diverse real-world applications, such as DBMS, web servers, AI workload, and Hadoop jobs, shows that vSMT-IO can improve I/O throughput by up to 88.3% and CPU throughput by up to 123.1%.

1.2.3 JUPITER: CPU Performance Isolation with High Efficiency for Multi-threaded Workloads in Time-Shared Clouds

The performance of multi-threaded applications is highly vulnerable to the time-sharing of CPUs, and coscheduling is an effective method to reduce such vulnerability. However, when there are two or more multi-threaded applications that are time-sharing CPUs, such as those in virtualized clouds, the effectiveness of existing coscheduling approaches is seriously limited. This causes a few performance issues compromising the Qualify of Service (QoS) for multi-threaded applications, such as the bad performance, the unfair performance penalty, low CPU utilization, and the sensitivity of the performance to other applications in the system.

Coscheduling becomes ineffective due to three reasons. 1) there lacks a mechanism to reduce/resolve the conflicting demands of coscheduling the threads from different applications on the shared resources; 2) there lacks a mechanism to reduce adverse effects of coscheduling without sacrificing its effectiveness; 3) existing coscheduling approaches cannot deal with dynamically changing workloads.

The dissertation proposes JUPITER that solves the above problems based on three ideas: 1) enforcing asymmetric coscheduling; 2) reducing the aggressiveness of coscheduling as long as application performance is not degraded; 3) effectively combining coscheduling with the leaky bucket technique. We implement JUPITER in KVM and conduct extensive experiments with diverse real-world applications to compare JUPITER with three existing variants of coscheduling. Experiments show that JUPITER achieves significantly and consistently better QoS and system-wide performance than the other systems under both homogeneous and heterogeneous workloads.

1.3 Structure of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 presents the vMIGRATER system, a simple, fast and transparent system to greatly mitigate I/O inactivity for multi-threaded workloads in cloud computing. Chapter 3 describes the vSMT-IO system, an efficient SMT-aware virtual CPU scheduler to improve multi-threaded performance in cloud computing. Chapter 4 introduces the JUPITER system, an effective framework with high efficiency to improve multi-threaded QoS in cloud computing. Chapter 5 concludes.

CHAPTER 2

VMIGRATER: EFFECTIVELY MITIGATING I/O INACTIVITY IN VCPU SCHEDULING

2.1 Introduction

To ease management and save energy in clouds, multiple virtual machines (VMs) are often consolidated on a physical host. In each VM, multiple virtual CPUs (vCPUs) often time-share a physical CPU core (aka., pCPU). The virtual machine monitor (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.

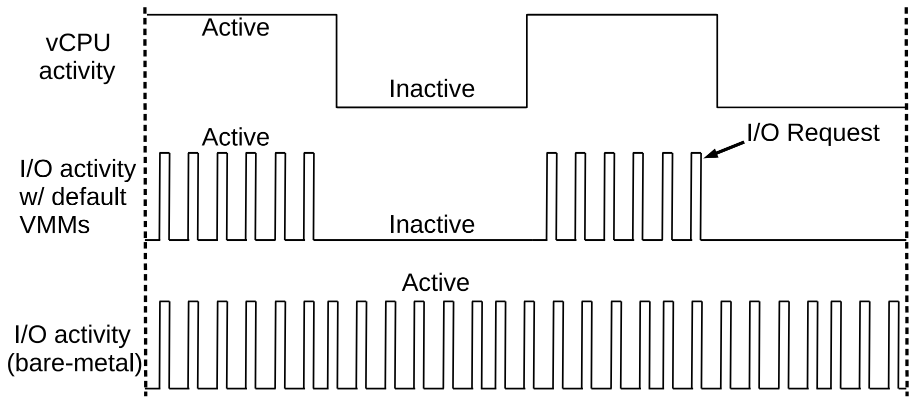


Figure 2.1 I/O inactivity.

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 2.1. 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 recommended by VMware [14]. We confirm this in our evaluation (§2.7).

The I/O inactivity problem becomes even more pronounced when I/O requests are supposed to be processed by fast storage devices (SSDs). Usually, a vCPU keeps active during I/O requests, so it can quickly process them. For example, if a computation task and an I/O task run on the same vCPU, when the I/O task issues a read request and waits for the request to be satisfied, the computation task is switched on. At this moment, the vCPU is not idleand descheduled; 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 by orders of magnitudes.

Worse, the I/O inactivity problem causes the I/O scheduler running in the host OS to work extremely poorly. To fully utilize storage devices, based on the latencies of I/O devices, system designs usually 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. Moreover, non-work-conserving I/O schedulers [15] usually hold the I/O resource and wait a period for the next request from the same I/O task (refer to §2.2.2). By serving the requests from the same task continuously, which shows better locality than the requests from different tasks, such I/O schedulers [15] 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 (confirmed in our evaluation).

Last but not least, the I/O throughput of a VM can be "capped" by its amount of CPU resources. 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 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 two VMs.

All these problems share the same root cause, I/O inactivity, but existing works mainly mitigate vCPU inactivity and ignore this root cause. Existing works mainly take two approaches: 1) shortening vCPU time slices (vSlicer [16]); and 2) assigning higher priority for I/O tasks running on active vCPUs (xBalloon [17]). Unfortunately, the vCPUs with both approaches can often be inactive, and fairness issues among VMs in a host can easily occur.

Our idea to mitigate I/O inactivity is that each VM often has active vCPUs, and we can efficiently migrate I/O tasks to these 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 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.

The dissertation presents vMIGRATER, a user level tool working in each VM. It is transparent as it does not need to modify an application, VM, VMM, or OS. 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 incurs a small overhead to applications due to 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' load in the same VM are well balanced. By keeping I/O bound tasks mostly active, vMIGRATER greatly reduces I/O inactivity, making applications run as the bare-metal curve in Figure 2.1.

vMIGRATER addresses three practical challenges. First, it needs to determine which task should be migrated. vMIGRATER migrates I/O bound tasks running on the vCPUs that will be inactive. To identify I/O bound tasks, vMIGRATER uses an event-driven model to collect I/O statistics and to detect I/O bound tasks within microseconds.

Second, vMIGRATER needs to determine when the I/O bound task should be migrated. vMIGRATER migrates the I/O bound task when the vCPU running this task will be inactive so that the task can continue to be active. vMIGRATER monitors each vCPU's time slice and use the length of previous time slice to predict the current one.

Third, vMIGRATER needs to decide where the task should be migrated to. Based on the collected time slice and I/O task information, vMIGRATER migrates to-be-descheduled I/O tasks to the active vCPUs according to the vCPUs' current load. An active vCPU with heavier load will be assigned fewer I/O tasks to reduce the overhead.

The dissertation implemented vMIGRATER in Linux and evaluated it on KVM [18] with a collection of micro-benchmarks and 7 widely used or studied programs, including small programs (sequential, random and bursty read) from SysBench [19], a distributed file system HDFS [20], a distributed database Hbase [21], a mail server benchmark PostMark [22], a database management system LevelDB [23], and a document-oriented database program MongoDB [24]. Evaluation shows that: 1) vMIGRATER is fast. Compared to default Linux KVM (vanilla), vMIGRATER improves all applications' throughputs up to 4.42X. vMIGRATER is 1.84X to 3.64X faster than both vSlicer and xBalloon. 2) vMIGRATER is scalable. Compared to vanilla, vMIGRATER improves all applications' throughputs from 1.72X to 4.42X as the number of shared VMs increase from 2 to 8. 3) vMIGRATER makes the I/O Scheduler in the host OS fair. Compared to vanilla, when two vCPUs share one pCPU, I/O bandwidth of each I/O task is almost the same, reducing unfairness between VMs by 6.22X.

The contribution contains two parts. First, we took the first step to quantify the severity of I/O inactivity in VMs. Second, we built vMIGRATER, a simple and practical user-level scheduler, which greatly improves the throughput of applications in virtualized systems.

2.2 Background and Motivation

In this section, we first introduce vCPU descheduling, I/O scheduler. Then, we explain the three performance problems caused by I/O inactivity in virtualized systems.

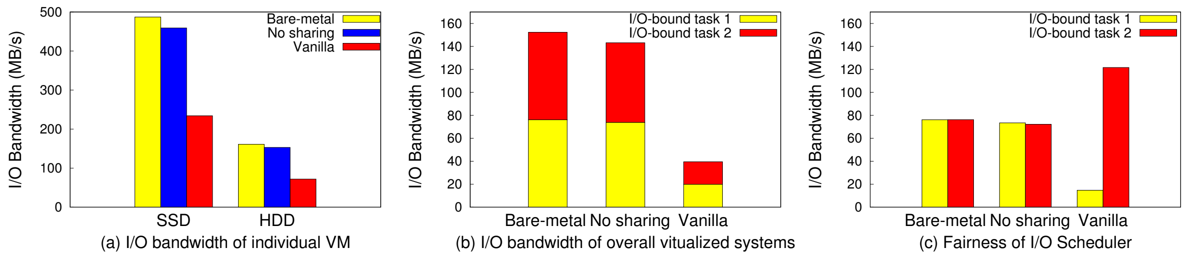


Figure 2.2 Three problems caused by I/O inactivity. "Bare-metal" means physical server; "No sharing" means one VM run on one host; "Vanilla" means two VMs run on one host.

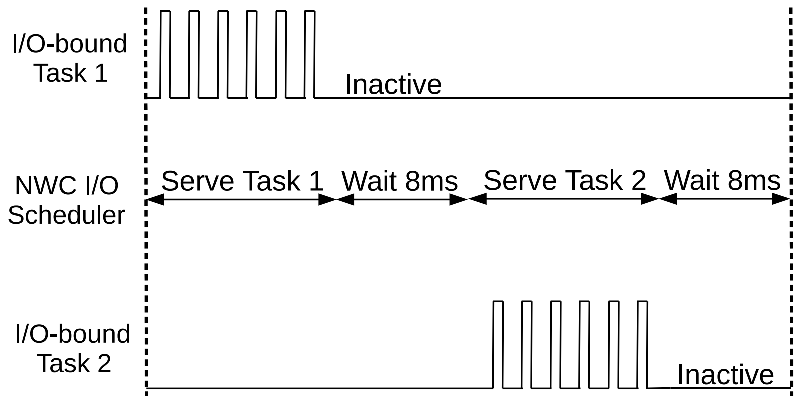


Figure 2.3 Explanation of performance degradation caused by Non-Work-Conserving (NWC) I/O scheduler in VMM. The wait time is wasted.

2.2.1 vCPU (De)scheduling

In virtualized systems, CPU sharing is widely adopted to improve resource utilization and performance isolation. CPU sharing allows many vCPUs to share one pCPU. Hypervisor schedules these vCPUs onto the pCPU. Hypervisor is often required to time-share the pCPU among these vCPUs, and it may deschedule a vCPU belonging to one VM in favor of a vCPU belonging to another VM. For instance, KVM uses completely fair scheduler (CFS) [25; 17] to schedule vCPUs onto pCPUs. CFS uses virtual runtime (vruntime), which tracks how much time a vCPU has spent running on pCPU, to schedule these vCPUs. CFS maintains a red-black tree-based runqueue, which sorts vCPUs based on their vruntimes, and always schedules the vCPU with the least vruntime. This design enforces fair allocation of vCPUs' time slices.

2.2.2 I/O Scheduler

I/O tasks are scheduled by the I/O scheduler in the VMM layer to access I/O devices for processing I/O requests. There are two types of I/O scheduler: work-conserving [26; 27] and non-work-conserving [28; 29]. The work-conserving I/O scheduler must choose one I/O request to serve if there is any pending request.

The non-work-conserving I/O scheduler, such as the anticipatory scheduler (AS) [29] and Completely Fair Queuing (CFQ) [30], waits a short period for the same task's I/O requests before switching to serve another task. This is because the same task's next request might be close to the disk head so the I/O scheduler may be worthwhile to wait. CFQ aims to fairly distribute disk time among I/O-intensive threads. As CFQ allows the disk to be idle, waiting for future requests, it also belongs to non-work-conserving type. In virtualized systems, non-work-conserving I/O scheduler (e.g., CFQ) is often used for performance and fairness benefits.

2.2.3 Problems Caused by I/O Inactivity

In this section, we show that Linux suffers a severe performance degradation when running as a virtualized system because of I/O inactivity. We use SysBench [19] to test I/O throughput in three settings. In the *Bare-metal* setting, SysBench runs on one host; In the *No sharing* setting, SysBench runs in one VM on one host (the VM and the host have the same number of cores); In the *Vanilla* setting, two VMs run on one host (two vCPUs share one pCPU). As such, the virtualized system in VM was active for a certain period and inactive for another period. Each VM was configured with four vCPUs and the same I/O bandwidth in KVM [18]. We use CPU workload from SysBench [19] as the compute-bound task co-running with the I/O-bound task. The setting in this section eases the analysis of the research problems, and the real workloads with common settings in the evaluation part (§2.7) show that these research problems are more severe.

Figure 2.2 shows the three performance issues caused by I/O inactivity in virtualized systems.

Figure 2.2 (a) shows that the I/O throughput problem caused by I/O inactivity. In this experiment, the two shared VMs are each allocated 50% pCPU resource, and only one VM has I/O bound task. The result shows that no sharing has roughly the same throughput as bare-metal but the throughput of vanilla degrades 49.8% due to I/O inactivity. Figure 2.1 explains the reason. In each VM, the vCPU is active for a time period and inactive for the same amount of time. This causes many I/O inactivity periods in the VM which runs the I/O-bound task.

Figure 2.2 (b) shows that when two VMs run I/O tasks in parallel, the total throughput degrades even more significantly. The total throughput drops by 72.1% compared to bare-metal and no sharing. Figure 2.3 explains the reason. The non-work-conserving I/O scheduler in the VMM serves one I/O-bound task for a short period, and waits for 8ms once the VM becomes inactive. After the wait period, the I/O scheduler changes to serve another I/O-bound task. The frequent change between tasks incurs the overhead of frequent disk seek, and the wasted waiting time also contributes to the inefficiency of the I/O scheduler.

Figure 2.2 (c) shows that non-work-conserving I/O scheduler of the VMM is not fair to VMs with different CPU resources. The two shared VMs are allocated with the same I/O bandwidth but with 20% and 80% CPU capacity respectively. The I/O scheduler in bare-metal and no sharing offer the same I/O bandwidth to I/O tasks. However, in the

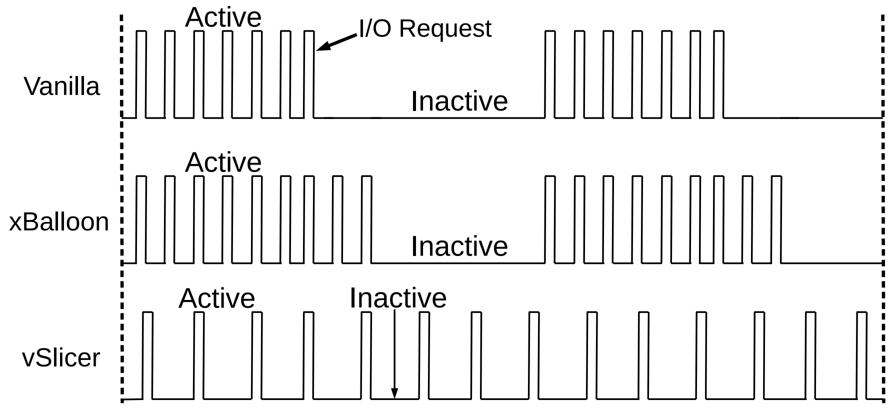


Figure 2.4 The workflow of vanilla, xBalloon and vSlicer. xBalloon and vSlicer still experience frequent I/O inactivity periods.

vanilla setting, I/O scheduler allocates VM2 5.8X I/O bandwidth than VM1. Figure 2.5 explains the reason of the unfairness. Since VM1 is only allocated much less CPU capacity than VM2, it experiences much longer I/O inactivity periods. As a result, I/O scheduler serves VM2 for much longer time than VM1, causing unfairness.

There are two types of existing works. The first approach [31; 32; 16] is to shorten time slices (e.g., vSlicer) for vCPU to process I/O requests more frequently. As shown in Figure 2.4, vSlicer reduces the length of each vCPU inactivity period so that the next I/O request can be processed with a shorter waiting time. However, the I/O inactivity period still exists and degrades the performance. Moreover, vSlicer incurs high context-switch overhead due to smaller time slices.

The second approach [33; 17] is to assign higher priority (e.g.. xBalloon) for I/O tasks running on active vCPUs. As shown in Figure 2.4, this approach gives more time to I/O bound tasks to improve the I/O throughput. The vCPUs are still descheduled so the I/O inactivity period still exists. Besides, this approach reduces the execution time of co-location compute bound tasks and degrades overall performance.

2.3 Overview

In this section, we present vMigrater's design goals (§2.3.1) and architecture (§2.3.2).

Table 2.1 Compare vMigrater with Related Systems.

Systems	vMigrater	vSlicer	xBalloon
Transparent	✔	✖	✖
Performance	✔	✖	✔
Scalability	✔	✔	✖
Fairness	✔	✖	✖

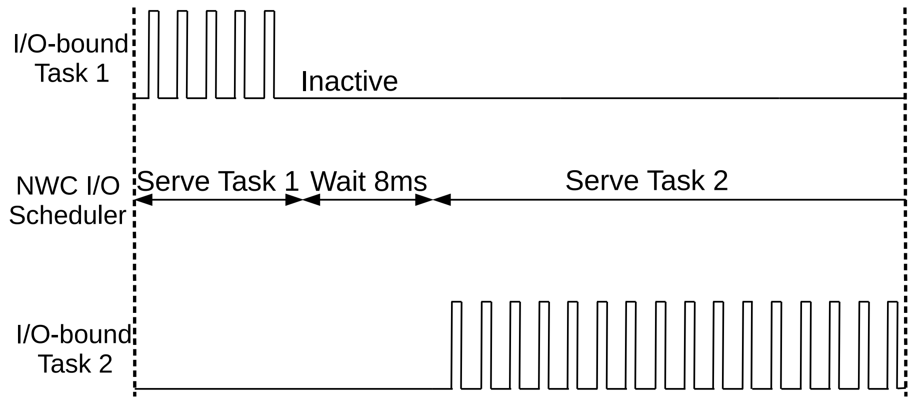


Figure 2.5 Explanation of unfairness of Non-Work-Conserving (NWC) I/O scheduler in VMM. The I/O scheduler serves task 2 with a much longer time.

2.3.1 Design Goals

Compared with related systems (as shown in table 2.1), vMIGRATER aims at three design goals. First, vMIGRATER should be transparent. The reality in virtualization environments is that VM and VMM source may not be available to modify. To support the broadest possible range of applications and virtualized platforms with the smallest possible barrier to entry, the design of vMIGRATER should not require to modify any part of applications, VM and VMM. Second, vMIGRATER should promise high performance for applications in VMs. Even though I/O-bound workloads run on shared vCPUs in virtualized systems (such as libOS VMs [34; 35]), they can achieve almost the same I/O bandwidth as they are running in physical machine. Third, vMIGRATER should make the I/O scheduler in the host OS maintain fairness [36; 37].

We choose to design vMIGRATER in the user space of guest OSes because it is general and transparent to applications, guest and host OSes, and devices. If we design vMIGRATER in the hypervisor layer, it may need to instrument guest and host OSes and is not a general approach.

2.3.2 vMIGRATER Architecture

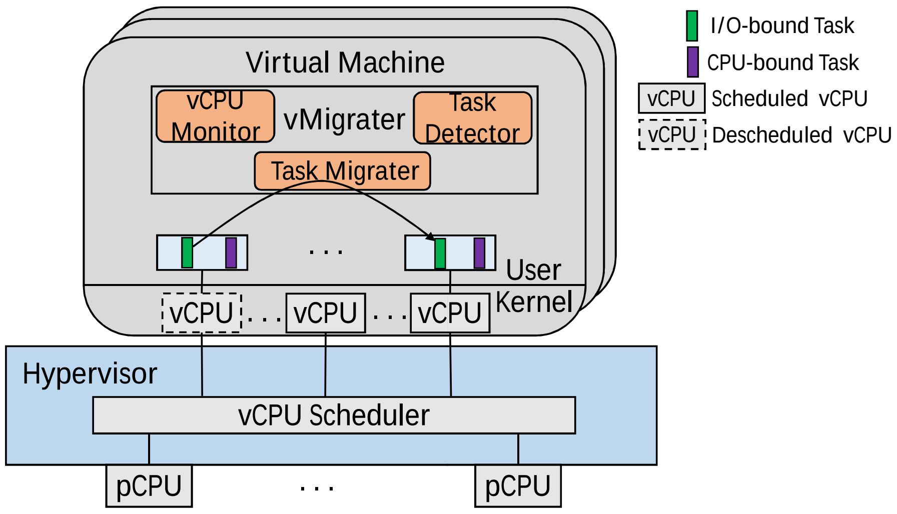


Figure 2.6 vMIGRATER Architecture.

Figure 3.2 shows vMIGRATER's architecture with three key components: vCPU Monitor, Task Detector and Task Migrater. vCPU Monitor and Task Detector collect the information of vCPUs and I/O tasks respectively. Based on this information, Task Migrater makes migration decisions and conducts the migrations. All these three components cooperate with each other to migrate I/O tasks to active vCPUs to mitigate I/O inactivity.

vCPU Monitor (§2.4.1) monitors vCPUs' time slices. It monitors the start and end of time slices for each vCPU and use the length of previous time slice to predict the current one.

Task Detector (§2.4.2) detects I/O-bound tasks within micro-seconds. It uses an event-driven model to collect the time each task uses for processing I/O requests within one interval.

Task Migrater (§2.4.3) migrates I/O-bound tasks to active vCPUs. Task Migrater knows which vCPU is active according to the information from vCPU Monitor and which task is a I/O-bound task with the information from Task Detector. Once Task Migrater finds the I/O-bound tasks located on the vCPUs that will be inactive soon and migrates the tasks to active vCPUs in a globally balanced way.

2.4 The vMIGRATER Runtime System

This section presents vMIGRATER's runtime system. We first introduce how vMIGRATER monitors vCPUs' time slices accurately (§2.4.1). Then, we explain how vMIGRATER fast detects I/O-bound tasks (§2.4.2), and how vMIGRATER migrates I/O-bound tasks efficiently (§2.4.3). At last, we present the performance analysis of vMIGRATER (§2.4.4).

2.4.1 Monitoring vCPUs' Time Slices

vMIGRATER runs one vCPU Monitor thread on each vCPU in the VM. Figure 2.7 shows how vCPU Monitor works. The vCPU Monitor thread sleeps periodically in order to reduce the overhead to co-running applications. On each wakeup, it compares the elapsed time against its sleep time to determine whether the vCPU has been descheduled. For instance in Figure 2.7 when the vCPU Monitor wakes up at t_6, it finds the elapsed time is much longer than the sleep time, so it determines that it has been descheduled and a new time slice starts. Then, it uses the time difference between t_2 and t_5 to calculate the time slice.

vCPU Monitor has three properties. First, vMIGRATER uses previous time slice length to predict the current time slice length. Our key observation is that shared vCPUs' time slices are almost stable when I/O- and compute-bound tasks are running concurrently in the VM. Intuitively, a workload often runs stably during one time period. In our evaluation (see 2.7.2), we show that when `Apache Hadoop` system runs in two VMs sharing one host, the time slices are almost stable.

Second, vMIGRATER predicts a vCPU which is to be active right now. When the remaining time slices of active vCPUs in a VM are not long, vMIGRATER migrates I/O bound tasks to the vCPU which is to be active. In this way, the number of migration decreases and the time cost reduces correspondingly. vMIGRATER monitors the time length of a vCPU has been descheduled(e.g., $t_6 - t_5$ in Figure 2.7), which can be used to check whether the vCPU is to be re-scheduled.

Third, the length of sleep time may affect vMIGRATER's performance. If the length is too long, the deviation of vMIGRATER measured time slice is big because it cannot accurately determine the start and end of a time slice (tens of ms). If the length is too short, vCPU Monitor thread wakes up and sleep frequently (preemption context switches, tens of ns). Therefore, vMIGRATER uses a general sleep period of 300 μs, which is in the middle of the above two constants and works well for all applications in our evaluation.

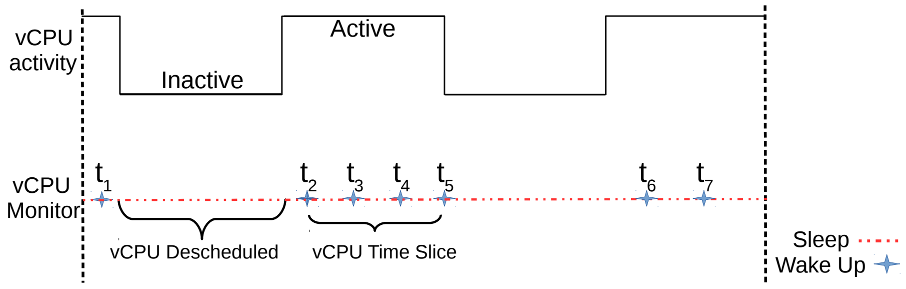


Figure 2.7 vCPU Monitor workflow.

2.4.2 Detecting I/O Bound Tasks

vMIGRATER needs to quickly detect I/O-bound tasks for migration. However, traditional methods (e.g., Linux top [38] and iotop [39]) to detect tasks' I/O utilization is too coarse-grained. The detect duration is often as long as at least one second. We need a faster frequency to detect I/O bound tasks. For instance, the time for SSD to handle 1MB sequential read is only 1ms.

We propose an event driven solution for fast detecting I/O-bound tasks. When an I/O request is sent to the block device, it will trigger an I/O event. vMIGRATER monitors the I/O events and collects the time spent to process these events. vMIGRATER calculates the fraction of I/O processing time in one period (several milliseconds) and determines whether the task is I/O bound. This method is flexible, accurate and fast because we can monitor tasks in a very short time period.

During our implementation, we found a bug [40] in Linux Kernel about collecting synchronous I/O delay time. We submitted a patch to the linux kernel's mailing list and are waiting for response.

2.4.3 Migrating I/O Bound Tasks

Task Migrater uses the information provided by vCPU Monitor and Task Detector to make migration decisions. First, Task Migrater needs to decide which I/O task should be migrated. This is determined by two factors. One factor is Task Migrater needs to find the I/O bound tasks whose vCPUs are to be descheduled. These I/O bound tasks should be migrated immediately to avoid inactivities. Another factor is that Task Migrater needs to find I/O bound tasks with higher I/O load because they affect applications' throughput largely. Task Migrater also supports user-defined priority to improve the performance of critical applications.

Second, Task Migrater needs to decide which vCPU should I/O bound tasks be migrated to. A naive approach is to migrate I/O tasks to the vCPU with the longest remaining time slices. However, this method has two problems: (1) the vCPU might be overloaded, greatly affecting performance of co-running compute tasks; (2) the I/O tasks cannot make progress concurrently. Task Migrater migrates I/O tasks to vCPUs in a globally balanced way. Task Migrater gives vCPUs different weights according to their remaining time slices and migrates I/O tasks to the vCPUs according to their I/O load level. For instance, the I/O task with the largest load level is migrated to the vCPU with highest weight. The migration mechanism effectively improves the throughout of applications because it distributes I/O intensive tasks among active vCPUs in a simple and efficient way.

2.4.4 Performance Analysis

Equation (1) shows the performance speedup of vMIGRATER. For simplicity, we assume each VM has at least one active vCPU at any given time. T_{pm} is execution time of one I/O task on dedicated pCPUs without sharing; N is the number of shared vCPUs on one pCPU; $N_{migrate}$ is the number of migration triggered by vMIGRATER; C_{avg} is the average time cost for each migration.

The numerator of equation (1) means the total execution time with vanilla (default sharing). Since the I/O task only runs 1/N time, the total execution time is roughly N times of T_{pm}. The denominator means the total execution time with vMIGRATER. The I/O task runs continuously with the only overhead from migration cost.

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

In an optimal scenario, vMIGRATER migrates an I/O bound task with a minimum number, N_{min}, which means the I/O task can always run from the rescheduled time point of each active vCPU. T_{ts} is the time slice of one vCPU in the shared VM. We have:

$$T_{pm} = T_{ts} \times N_{min} \tag{2}$$

After substituting for T_{pm} from (2) to (1), we get:

$$Speedup_{vMigrater} = \frac{N}{1 + \frac{N_{migrate} \times C_{avg}}{N_{min} \times T_{ts}}} \tag{3}$$

Equation 3 defines the speedup in terms of N and $\frac{N_{migrate} \times C_{avg}}{N_{min} \times T_{ts}}$; N and T_{ts} are constants. We denote $\frac{N_{migrate} \times C_{avg}}{N_{min} \times T_{ts}}$ as $P_{vMigrater}$, so the speedup is mainly in terms of $P_{vMigrater}$. If $P_{vMigrater}$ is very small, the speedup is nearly N times. Our evaluation confirms that the speedup of vMIGRATER matches $Speedup_{vMigrater}$ (see §2.7.2).

2.5 Implementation Details

In order to implement vCPU Monitor accurately, vMIGRATER needs to call the system timer. However, we find that system time clock is inaccurate due to CPU sharing [17]. The traditional timer interrupt is triggered by each shared vCPU but the interrupt may be ineffective once the vCPU is descheduled. In vMIGRATER, we use the clock source (CLOCK_MONOTONIC [41]) updated by global timer interrupts which can be triggered by any vCPU in the VM. From our experiments, we find the clock source updated by global timer interrupts is much accurate. The reason is that in a multi-vCPU VM, the possibility of descheduling all the vCPUs is low.

vMIGRATER leverages BCC [42; 43] to implement fast Task Detector. BCC is a toolkit for creating efficient kernel tracing and manipulation programs. BCC has been supported by Linux Kernel and provided a way to filter out I/O tasks once there are I/O events handled by I/O devices. We implemented our fast I/O tasks detection module by modifying source codes of BCC.

We implement Task Migrater with two mechanisms, PUSH and PULL. PUSH means a vCPU, I/O bound tasks are running on, can push these tasks to active vCPUs which have longer remaining time slices. PUSH happens when the vCPU finds that its remaining time slice is lower than a threshold which is self-defined. PULL means a vCPU with a longer time slice can pull I/O bound tasks on to itself. This usually happens when a vCPU is active just now. vMIGRATER needs PULL because the I/O bound task may be on the de-scheduled vCPU. The time slice of the vCPU may not be very stable all the time. When the time slice is unstable, the I/O task may not be pushed on time so we implement PULL to solve this problem.

2.6 Limitations

vMIGRATER has two limitations. First, when vMIGRATER runs in a VM, the performance of an application in the VM may drop temporarily when the application's workload changes suddenly. Our evaluation (see §2.7.3) shows that, when the number of clients for HDFS increased from 16 to 32, HDFS's throughput dropped by 45.6% for 1.3 seconds and then went back to the peak throughput immediately. The reason is that the sudden changing workload makes time slices of some vCPUs not stable, and vMIGRATER needs some time to precisely re-estimate the time slices of vCPUs and then migrate I/O bound tasks.

Second, vMIGRATER mainly aims to mitigate the performance degradation caused by disk (HDD or SSD) I/O inactivity periods in VMs, and it is not designed to handle network I/O. Comparing to disk I/O, network I/O is much more sparse, and we have not found that vMIGRATER affects the performance of network I/O in our evaluation.

2.7 Evaluation

Our evaluation is done on a DELL^{TM} PowerEdge^{TM} R430 server with 64GB of DRAM, and one 2.60GHz Intel Xeon E5-2690 processor with 12 cores, and 1TB HDD or SSD separately. All VMs (unless specified) have 12 vCPUs and 4GB memory. The VMM is KVM [18], and both the host OS and guest OS are Ubuntu 16.04. The time slice of vCPU is 11ms (recommended by Red Hat [44]). The I/O scheduler in VMM is CFQ with the wait time of 8ms, recommended by Red Hat [45; 15].

We evaluate vMIGRATER on a collection of micro-benchmarks (SysBench [19] sequential read and random read; bursty read is implemented by us) and 7 widely used applications (`HDFS` [20], `LevelDB` [23], `MediaTomb` [46], `HBase` [21], `PostMark` [22], `Nginx` [47] and `MongoDB` [24]). To be close to real-world deployments, `HDFS` and `HBase` are ran with `MapReduce` [48] because they belong to `Apache Hadoop` system and are ran together. `PostMark` is ran with `ClamAV` (antivirus program) [49] because `PostMark` is a mail server benchmark and needs an antivirus program. `LevelDB` and `MongoDB` are deployed as the back-end storage of Spark [50].

Table 3.4 shows our workloads. For `Apache Hadoop` and `Spark` systems, we co-run the `TeraSort` benchmark [51] as the compute-bound task. `ClamAV` is a compute-bound program, which runs with the `PostMark` mail server benchmark. For `Nginx` and `MediaTomb`, transcoding and watermarking launch concurrently I/O and compute-bound tasks.

Table 2.2 Seven Applications and Workloads.

Application	**Workload**
`HDFS`	Sequential read 16GB with `HDFS TestDFSIO` [51].
`LevelDB`	Random scan table with db_bench [52].
`MediaT`	Concurrent requests on transcoding a 1.1GB video.
`HBase`	Random read 1GB with `HBase PerfEval` [51].
`PostMark`	Concurrent requests on a mail server.
`Nginx`	Concurrent requests on watermarking images [53].
`MongoDB`	Sequential scan records with YCSB [54].

All the experiments are conducted on SSD except that §2.7.4 (fairness of I/O scheduler) evaluates vMIGRATER on HDD. We use HDD for the fairness evaluation because HDD often uses CFQ.

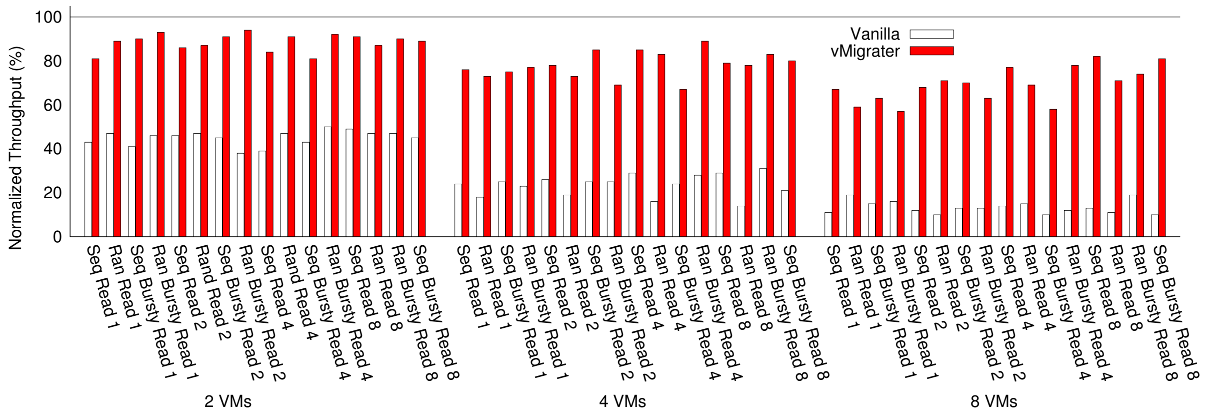


Figure 2.8 SysBench microbenchmarks, all VMs run the same microbenchmark, normalized to "no vCPU sharing" (i.e., only one 12-vCPU VM runs on a 12-pCPU host). "Vanilla" means the default Linux KVM. "App X" means the microbenchmark application with an X number of I/O tasks. "Y VMs" means a Y number of VMs are running on one physical host.

We compare vMIGRATER with two related systems: xBalloon [17] and vSlicer [16]. Because neither of them is open source, we implement both of them according to their papers. This section focuses on four questions:

§2.7.1: Is vMIGRATER easy to use?

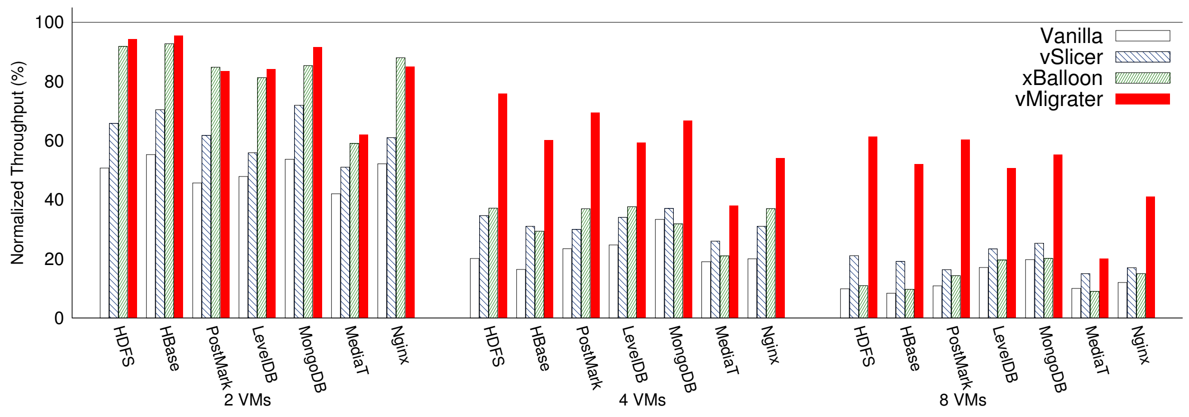


Figure 2.9 Real applications, all VMs run the same application, the average throughput of all VMs on one host, normalized to "no vCPU sharing".

§2.7.2: How much can vMIGRATER improve performance? How does it scale to the number of shared vCPUs per pCPU? How faster is vMIGRATER than prior systems?

§2.7.3: What is vMIGRATER's performance under varied workloads?

§2.7.4: Does vMIGRATER achieve fairness for the I/O scheduler?

2.7.1 Ease of Use

All 7 real applications we evaluated are able to be transparently plugged and played with vMIGRATER without any modification. When we evaluate these applications, vMIGRATER runs in the user-level of the guest OS and also does not need to change any part of the VM and VMM. vMIGRATER can support general hypervisors transparently.

2.7.2 Performance and Scalability

Figure 2.8 and 2.9 show that vMIGRATER improves the overall throughput significantly. All results are normalized to "no sharing". The micro-benchmark or application runs in each VM to test the overall throughput. On average, for micro-benchmarks, vMIGRATER makes them run 5.31X faster than vanilla. For real applications, vMIGRATER makes them run 4.42X faster than vanilla.

vMIGRATER greatly improves application performance due to two reasons. First, vMIGRATER effectively avoids I/O inactivity by migrating I/O-bound tasks between active vCPUs (see table 2.4). Second, vMIGRATER decreases the wait periods and improves locality for the I/O scheduler in the host OS.

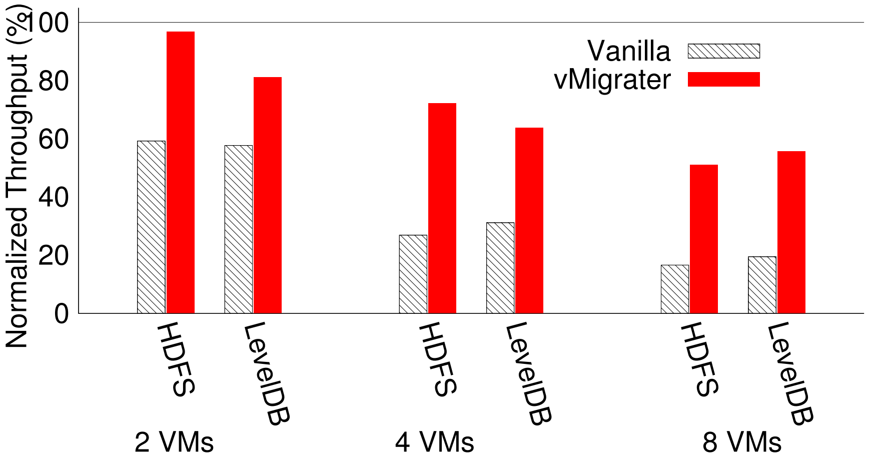


Figure 2.10 Throughput of only one VM running the application with vMIGRATER. Normalized to "no vCPU sharing", while the other VMs ran compute-bound task (i.e., IS from NPB) without vMIGRATER.

Figure 2.10 shows that vMIGRATER improves the throughput of an individual VM. All results are normalized to no sharing. One VM runs the application with vMIGRATER, and co-running VMs run compute-bound tasks (IS from NPB [55]) without vMIGRATER. On average, vMIGRATER makes the individual VM run 3.07X faster than vanilla.

Table 2.3 and 2.4 explain why vMIGRATER's performance is higher. Table 2.3 confirms the performance modeling of vMIGRATER's speedup. vMIGRATER incurs small

Table 2.3 vMIGRATER Performance Analysis for Seven Applications

Application	$N_{migrate}$	N_{min}	C_{avg}	$P_{vMigrater}$	Speedup
`HDFS`	3363	3181	0.87ms	0.07	1.86
`LevelDB`	2154	2003	1.62ms	0.13	1.75
`HBase`	3454	3181	1.88ms	0.15	1.76
`MongoDB`	1545	1363	2.16ms	0.17	1.70
`PostMark`	5181	4818	1.11ms	0.09	1.82
`MediaTomb`	2454	1818	5.23ms	0.35	1.41
`Nginx`	4181	4090	2.17ms	0.22	1.73

overhead for migration, 1.63ms in average. Table 2.4 shows that vMIGRATER reduces the I/O inactivity periods up to 18.39X than vanilla.

Figure 2.8 and 2.9 show vMIGRATER's performance is scalable to the number of shared vCPUs per pCPU. For micro-benchmarks, vMIGRATER improves the throughput from 1.97X to 5.31X as the number of shared VMs on one host increased from 2 to 8. For 7 real applications, vMIGRATER improves the throughput from 1.72X to 4.42X when the number of shared vCPUs on one pCPU increases from 2 to 8.

vMIGRATER's performance speedup is not ideal for 8 (or 4) VMs sharing one host. This is because when the number of shared vCPUs per pCPU increases, the portion of active vCPUs in a VM decreases, so the migration cost of vMIGRATER also increases. Nevertheless, in this high sharing setting, vMIGRATER is already several times faster than vanilla, vSlicer and xBalloon.

Figure 2.9 shows vMIGRATER, vSlicer and xBalloon's throughput on seven applications. All results are normalized to no sharing. On average, vMIGRATER's throughput is 2.48X and 3.64X higher than vSlicer and xBalloon separately.

The three systems do not have high performance on `MediaTomb`. `MediaTomb` has multiple tasks and combines I/O and compute operations in each task. The migration overhead for the task of `MediaTomb` is bigger than I/O-bound task. xBalloon pauses VM to benefit I/O-bound task. If I/O and compute are mixed in one task, the throughput degrades. vSlicer makes a smaller time slice for processing I/O-bound task in time. It incurs much more context switches overhead for the compute part of this task.

To analyze vSlicer's throughput, we also look into `MediaTomb`, which has concurrent requests to read the video to the memory for transcoding. We find that vSlicer incurs much more context switches overhead because it uses a much lower time slice for vCPU. With vSlicer, I/O intensive tasks still have smaller I/O inactivity period caused by descheduling vCPUs (see table 2.4). vMIGRATER does not incur extra context switches and can avoid I/O inactivity periods efficiently.

Figure 2.9 shows that xBalloon has a slightly higher throughput than vMIGRATER when two VMs share one host. When four or eight VMs share one host, the throughput of vMIGRATER is much higher than xBalloon. Table 2.4 shows that when 4 VMs share one host, xBalloon incurs more I/O inactivity periods because xBalloon does not utilize the CPU resource of other active vCPUs in the same VM. xBalloon's paper [17] confirms that xBalloon incurs performance drop in the setting of 4 or 8 VMs per host.

Table 2.4 I/O Inactivity Periods (Seconds) for Seven Applications.

Application	Vanilla	vSlicer	xBalloon	vMigrater	Mini
`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

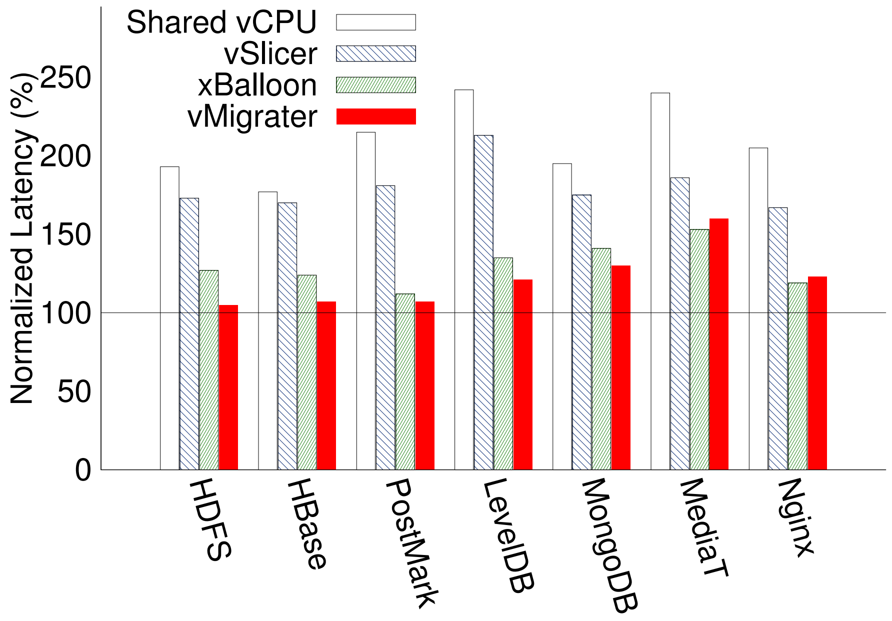


Figure 2.11 Response time normalized to "no vCPU sharing". Two 12-vCPU VMs share 12 pCPUs.

Figure 2.11 shows the response time of the three systems normalized to no sharing. For 7 applications, the response time of VMIGRATER and xBalloon is almost the same. Since each VM has 50% CPU resource, xBalloon has good performance (mentioned above). For `MediaTomb`, all three systems incur high response time because `MediaTomb` combines I/O and compute operations in one task.

Figure 2.12 shows three systems' overhead to co-running compute-bound applications in the same VM. xBalloon's overhead is much higher than VMIGRATER and vSlicer because xBalloon prioritizes I/O-bound tasks and delays compute-bound tasks. vSlicer's overhead is higher than VMIGRATER because it incurs much more context switches overhead for compute-bound tasks. Unlike xBalloon and vSlicer, VMIGRATER almost does not delay co-running applications (§2.4).

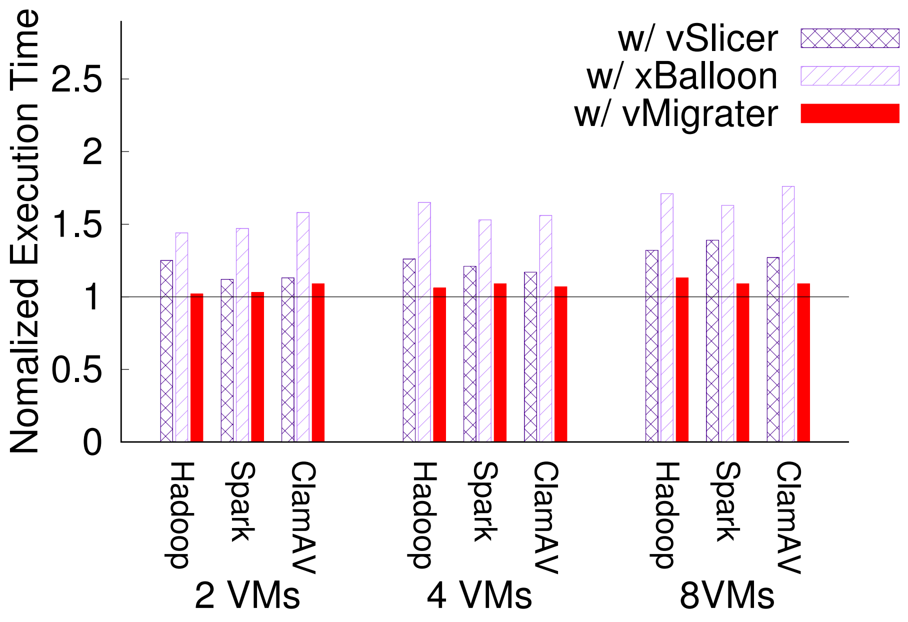


Figure 2.12 Execution time normalized to "Vanilla". "Hadoop" means each VM is running Hadoop standard TeraSort workload; "Spark" means each VM is running standard WordCount workload; "ClamAV" means each VM is scanning virus for the whole OS. Each VM has 12 vCPUs.

2.7.3 Robustness to Varied Workloads

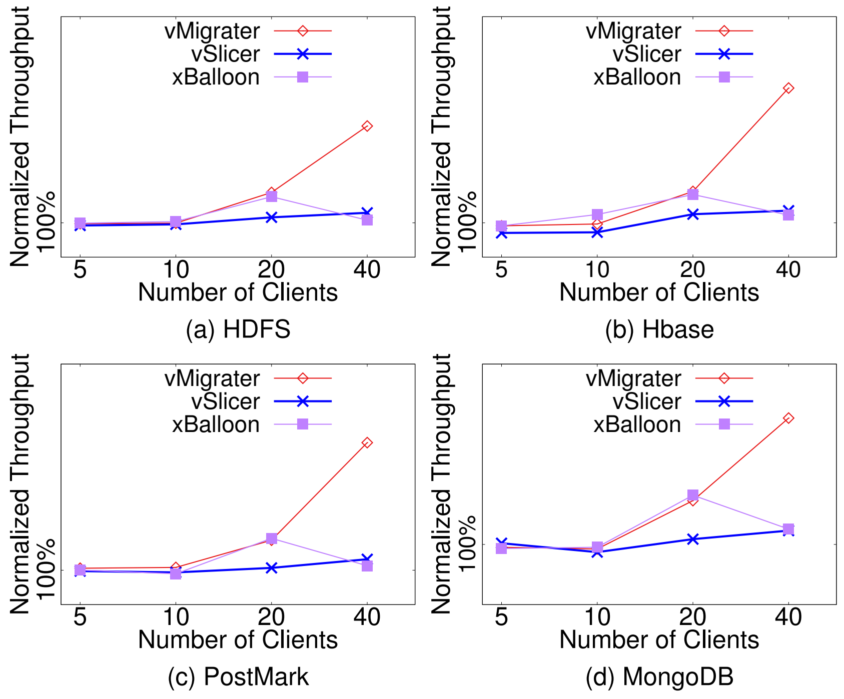


Figure 2.13 Throughput scalability on the loads of VMs, normalized to vanilla. Each client exhausts around 20% CPU resources; two 2-vCPU VMs share two pCPUs; The more concurrent clients, the faster vMigrater than vSlicer and xBalloon.

Figure 2.13 shows three systems' throughput on varied loads. When the number of clients is lower than 10, the throughput of vMigrater is almost the same as vSlicer and xBalloon because VMs are not shared. vMigrater is not started when there is no sharing. As the number of clients increased to 40, vMigrater outperforms other two systems significantly because vMigrater can efficiently avoid I/O inactivity periods by migrating I/O tasks to scheduled vCPUs. vSlicer and xBalloon do not work when vCPU is inactive. xBalloon has almost the same performance as vMigrater around 20 clients (each VM has 50% CPU resource). However, vMIGRATER is much more scalable than vSlicer and xBalloon when workloads increase.

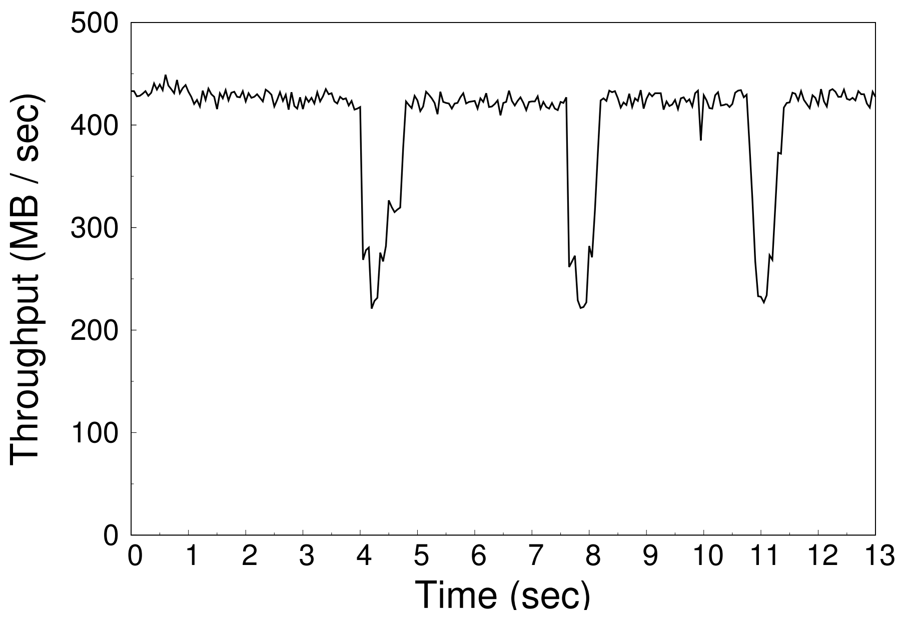


Figure 2.14 vMIGRATER's performance on handling the load change of vCPUs by adding clients dynamically. 8 clients at the time 0; each client exhausts 20% CPU resource; Two 2-vCPU VMs share two pCPUs.

Figure 4.14 shows the robustness of vMIGRATER on suddenly changing workloads. There are 8 clients at 0s, and VMs are not overloaded. At around 4s, 8s and 11s , 4, 8 and 16 more clients are added, vMIGRATER's throughput decreases to around 240MB/s, but it becomes stable (peak, around 430MB/s) again after a short period because vMIGRATER needs some time to precisely re-estimate the time slices of vCPUs and then mi- grate I/O bound tasks (§2.6).

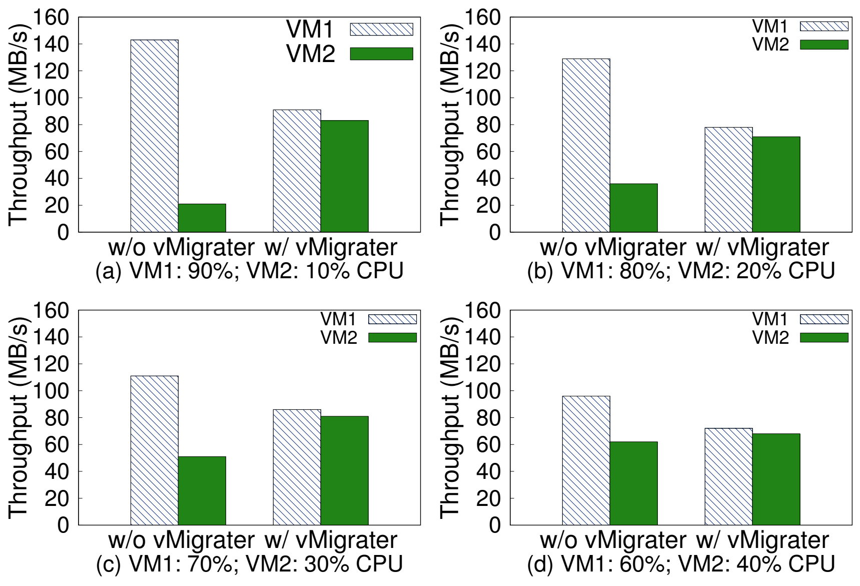


Figure 2.15 vMigrater improves the fairness of I/O Scheduler. Two 12-vCPU VMs share 12 pCPUs; each VM is allocated with different CPU resource but same I/O bandwidth.

2.7.4 Fairness for I/O Scheduler

Figure 2.15 shows the fairness of the VMM I/O scheduler among VMs. In Figure 2.15 (a), (b), (c) and (d), VM1's CPU resource decreases from 90% to 60%, and VM2's CPU resource increases from 10% to 40%. Each VM runs `TestDFSIO` (I/O-bound task) and `TeraSort` (compute-bound task) concurrently, and each VM is allocated with the same I/O bandwidth. Without vMigrater, `TestDFSIO` throughput is related to the CPU resource allocated to the VM, which shows vanilla hurts the fairness of the I/O scheduler in the host OS. With vMigrater, two VMs in each figure achieve roughly the same `TestDFSIO` throughput, which implies that vMigrater maintains fairness (roughly the same I/O bandwidth) for the two VMs.

2.8 Related Work

Shortening time slices. Many efforts focus on shortening time slices of vCPUs [32; 16; 31] for vCPU to process I/O requests more frequently. This solution has two obvious drawbacks: (1) the I/O inactivity period still exits and degrades I/O performance. (2) it suffers performance degradation because of frequent context switches [56; 57; 58]. (3) If the time slice is shorter than CFQ's wait time, even if one vCPU is scheduled, the CFQ may be still waiting for another inactive vCPU. The I/O requests issued from the scheduled vCPU still cannot be processed. C. Xu et al. [31] uses the same idea to reduce the delay of IRQ processing. These solutions require intensive modifications to both the VMM and guest OS kernel.

Dedicating CPUs. Dedicating CPUs [59; 60] aims to solve resource contention problem. This solution makes fewer vCPUs to share one pCPU to reduce contention. These systems are complementary to vMIGRATER because they focus on reducing the vCPU sharing, while vMIGRATER focuses on improving performance in the shared setting.

Task-aware Priority Boosting. Many existing systems [56; 33; 61; 17; 36; 62; 63; 64; 65; 66; 67; 68; 69; 70] focus on prioritizing latency-sensitive tasks to improve overall performance. Task aware VM scheduling [33] improves the performance of workloads by prioritizing I/O bound VMs.vMigrater differs task-aware VM scheduling mainly in two aspects: (1) task-aware VM scheduling prioritizes I/O bound VMs and promises CPU fairness among these VMs; vMigrater migrates I/O-bound tasks to active vCPUs to solve I/O inactivity problems and promises the fairness of I/O bandwidth among VMs. (2) task-aware VM scheduling works in the VMM layer and may need to change the source codes of host OSes; vMigrater works in the user space of guest OSes and does not need to change the source codes of applications, guest and host OSes.

Gleaner [56] identifies Blocked Waiter Wakeup (BWW) problem (eg, spinning, blocking, etc) due to the high overhead of idleness transitions in virtualized systems. It consolidates tasks on fewer vCPUs to reduce the number of vCPU state transitions. vBalance [61] presents that I/O interrupt is delayed due to CPU sharing. It re-maps I/O interrupt to an active VCPU for processing without delay. However, this work ignores that even if I/O interrupts are handled timely, I/O tasks still cannot make progress because vCPUs are descheduled.

xBalloon preserves the priority of I/O tasks by preserving CPU resource for I/O tasks. However, the vCPUs are still descheduled so the I/O inactivity period still exists. xBalloon and VMIGRATER are complementary to each other because xBalloon works best for VMs with single vCPU, while VMIGRATER is designed for multi-vCPU VMs.

2.9 Conclusion

This paper identifies the understudied problem of I/O inactivity in VMs. It presents VMIGRATER, a simple, fast and transparent system to greatly mitigate I/O inactivity.

CHAPTER 3

VSMT-IO: IMPROVING I/O PERFORMANCE AND EFFICIENCY ON SMT PROCESSORS IN VIRTUALIZED CLOUDS

3.1 Introduction

Simultaneous Multi-Threading (SMT), or Hyper-Threading (HT) on x86 processors, is widely enabled on most cloud infrastructures [71; 72; 73; 74]. For example, in Amazon EC2 [71], virtual instances can have their virtual CPUs (vCPUs) run on dedicated hardware threads or time-share hardware threads. With SMT, multiple hardware threads share the same set of execution resources in each core, such as functional units and caches. Thus, when enabled, SMT can effectively improve resource utilization and system throughput.

On SMT processors, CPU schedulers are critical for achieving high performance. To make optimal scheduling decisions, they must fully consider and leverage the performance features of SMT processors, particularly the intensive resource sharing between hardware threads. For example, intensive study has concentrated on symbiotic scheduling algorithms, which co-schedule the threads that can fully utilize the hardware resources with minimal conflicts on each core [75; 76; 77; 78; 79; 80].

Existing scheduling optimizations for SMT processors, including symbiotic scheduling and other enhancements in existing system software [81; 82; 83], mainly target computation-intensive workloads and aim to improve processor throughput. However, the techniques that can effectively and efficiently improve the performance of I/O-intensive workloads on SMT-enabled systems have not been paid enough attention. These techniques are particularly important when a system has both computation workloads and I/O workloads, and requires both high processor throughput and high I/O throughput.

To improve I/O workload performance, existing CPU schedulers generally use two techniques, polling [84; 85; 86] and boosting the priority of I/O workloads [17; 61; 87]. However, with these techniques, I/O workloads incur high overhead on SMT processors due to busy-looping and increased context switches, which can significantly reduce the performance of computation running on other hardware threads.

This problem is particularly significant and detrimental in clouds. In clouds, I/O workloads and computation workloads are usually consolidated on the same server to improve system utilization [17; 87; 88; 89; 90]. At the same time, virtualization is dominantly used in clouds, which causes busy-looping and context switches to incur higher overhead, because extra operations must be carried out to deschedule and reschedule virtual CPUs, as we will show in §3.2.

To control the overhead of polling and I/O workload priority boosting, existing system designs make trade-offs between the efficiency and the effectiveness of these techniques, which undermine the performance of I/O workloads. For polling, existing systems usually incorporate a short timeout to keep the busy-looping brief. For priority boosting, it has been a long-standing dilemma to make I/O workloads preempting the running workloads promptly or with some extra delay; to resolve this dilemma, Linux uses a scheduling delay parameter (tunable, usually a few milliseconds) as a knob to trade-off I/O workloads responsiveness and the increased context switch overhead.

Instead of improving the effectiveness-efficiency trade-off, the paper seeks a fundamental solution to the above problem. The key is a technique that can effectively improve the performance of I/O workloads with high efficiency. Our solution is motivated by the hardware-based design for efficient blocking synchronization on SMT processors [91]. With the design, blocking synchronizations can be finished efficiently without busy-looping or context switches. Specifically, the design allows a thread blocked at a synchronization point to free all its resources for other hardware threads to use, except for its hardware context; thus, when the thread is unblocked, it can resume its execution in a few cycles.

Our solution targets virtualized clouds and x86 SMT processors. It is built on a hardware-based blocking mechanism for vCPUs, named **Context Retention**. Context retention is implemented with Intel vCPU Monitor/`mwait` support [92]. With context retention, when a vCPU is waiting for an I/O event, its execution context can be held on a hardware thread without busy-looping involved; upon the I/O event, the vCPU can resume execution quickly without a context switch.

3.1.1 Technical Issues

While the rationale of the context retention mechanism is straightforward, maximizing its potential on improving performance needs to address three technical issues listed below. These issues arise mainly because context retention may be long time periods. Many I/O operations have long latencies in millisecond scale, and the latencies may further increase due to queueing/scheduling delays. To avoid context switches, the contexts of the vCPUs waiting for the finish of these operations need to be retained on hardware threads for the same amount of time.

First, uncontrolled context retention can diminish the benefits from simultaneous multithreading, because context retention reduces the number of active hardware threads on a core. This issue is particularly serious for x86 processors, which only implement 2-way SMT[1]. When a hardware thread is used for context retention, only one hardware thread remains for computation.

Second, context retention consumes the timeslice of an I/O workload, and thus reduces its timeslice available for computation. We found that, if not well controlled, context retention can even reduce the throughput of I/O workloads.

Third, due to context retention and burstiness of I/O operations [93], the resource demand of an I/O workload may vary dramatically on a hardware thread. This makes it a challenging task to improve processor throughput with existing symbiotic scheduling methods. To determine which workloads may make fast progress if scheduled on the same core, existing symbiotic scheduling methods periodically profile workload executions and make predictions based on the profiling results. Thus, these methods are effective only when the workload on each vCPU changes steadily. They must be substantially extended to handle I/O workloads.

3.1.2 Major Techniques

We implement our solution and address the above issues by designing the vSMT-IO scheduling framework. It has two major components. The **Long-Term Context Retention (LTCR)** mechanism is mainly to maximize I/O throughput with high efficiency.

[1]Though some Xeon Phi processors implement 4-way SMT, the paper targets 2-way SMT x86 processors because of their overwhelming dominance in clouds.

The **Retention Aware Symbiotic Scheduling (RASS)** algorithm is mainly to maximize processor throughput.

The LTCR mechanism mainly addresses the first two issues identified in Section 3.1.1. It holds the context of the vCPU waiting for an I/O event on a hardware thread for an extended time period. If the expected I/O event happens in this period, the vCPU can quickly resume and respond to the event. Otherwise, the vCPU is descheduled. The maximum length of the time period is carefully adjusted in a way that both processor throughput and I/O throughput can be improved.

With LTCR, the context of an I/O workload can be held for as long as a few milliseconds, which is more than 10x longer than the busy-looping timeout used in system software (sub-millisecond) [84; 85]. This makes LTCR capable of dealing with relatively high I/O latencies, which are associated with slow I/O operations (e.g., HDD accesses and SSD writes) or caused by various system factors (e.g., queueing/scheduling delay and SSD block erase). In contrast, polling is used only when I/O workloads interact with low latency devices, e.g., local network and NVMe devices [94; 86].

The RASS algorithm mainly addresses the third issue identified in Section 3.1.1. On each core, it classifies the vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs. It uses one hardware thread for running CPU-bound vCPUs and the other hardware thread mainly for I/O-bound vCPUs. In this way, the computation on the CPU-bound vCPUs can overlap to the greatest extent with the context retention periods on the other hardware thread. This effectively improves processor throughput, since CPU-bound vCPUs can take advantage of the hardware resources released due to context retention to make fast progress. RASS schedules CPU-bound vCPUs on both hardware threads only when I/O-bound vCPUs are not ready to run. In this case, RASS selects CPU-bound vCPUs based on the symbiosis between vCPUs (i.e., how well the vCPUs can share the hardware resources and make progress when co-scheduled).

With RASS, the first two issues identified in Section 3.1.1 can be further mitigated. LTCR mainly targets long context retentions. It limits the lengths of context retentions to mitigate the resource underutilization they cause and reduce the timeslice they consume. However, it cannot deal with the issues caused by relatively short context retentions. For these context retentions, RASS mitigates the resource underutilization issue (the first issue in Section 3.1.1) by overlapping computation and context retention; to mitigate the second issue, it helps ensure the supply of timeslice to I/O-bound vCPUs by running them on dedicated hardware threads with high priorities.

The paper makes the following contributions. First, the paper identifies the efficiency issues in existing CPU schedulers when they are used to improve I/O performance on SMT-enabled systems, and proposes a novel idea, context retention, to improve efficiency. Second, it identifies the issues in implementing the idea, and explores effective techniques to address these issues, including long term context retention and retention-aware symbiotic scheduling. Third, targeting virtualized clouds and x86 processors, the paper designs vSMT-IO to implement the idea and the techniques, and builds a system prototype based on KVM [95]. Forth, it has evaluated vSMT-IO with extensive experiments and a diverse set of 7 programs, including DBMS, web servers, AI workloads, and Hadoop jobs, and compared the performance of vSMT-IO with the vanilla system and widely-adopted enhancements. The experiments show that vSMT-IO can improve I/O throughput by up to 88.3% and processor throughput by up to 123.1%.

3.2 Background and Motivation

Targeting virtualized clouds, this section demonstrates the efficiency issues of existing schedulers in improving I/O performance on SMT-enabled systems. It first introduces these techniques, and experimentally verifies their inefficiency and the caused performance degradation (§3.2.1). Then, it explains why the issues are serious on virtualized platforms (§3.2.2).

3.2.1 Inefficient I/O-Improving Techniques

I/O-intensive applications are usually driven by I/O events. A pattern repeated in their executions is waiting for I/O events (e.g., queries received from network, or data read from disks), processing I/O events, and generating new I/O requests (e.g., responses to queries, or more disk reads). Thus, high I/O performance not only depends on fast and well-managed I/O devices to quickly respond to I/O requests. It also depends on the applications to promptly respond to various I/O events, such that new I/O requests can be generated and issued to I/O devices quickly.

Thus, CPU schedulers play an important role in improving I/O performance. To increase the responsiveness of I/O workloads to I/O events, existing schedulers use two general techniques — polling for low-latency I/O events and priority boosting for high-latency I/O events. With polling, an I/O workload waiting for an I/O event enters a busy loop (implemented with PAUSE on x86 processors) with a pre-set timeout. The workload keeps looping before it is interrupted upon the expected I/O event or is descheduled due to timeout. Thus, polling allows a workload to respond to I/O events with a minimal delay before timeouts. With priority boosting, upon an I/O event, the priority of the I/O workload is boosted, such that it can quickly preempt a running workload to respond to the I/O event.

On virtualized platforms, I/O workloads run on vCPUs; and vCPU scheduling becomes a key component affecting I/O performance. For vCPUs, polling may be implemented in guest OS kernel [96]. However, busy-looping in guest OS causes unnecessary VM_EXITs and extra overhead on x86 processors when Pause Loop Exiting (PLE) is enabled. Thus, recent designs (e.g., HALT-Polling [85]) usually implement polling at the VMM level. Priority boosting may be implemented by adjusting priorities explicitly [17] or by implicitly associating priorities with CPU time consumption. For example, Linux/KVM allows the vCPUs with lower CPU time consumption (e.g., I/O-bound vCPUs) to preempt the vCPUs with higher CPU time consumption [97; 17].

Though polling and priority boosting can improve the performance of I/O workloads, they are inefficient on SMT processors. The operations associated with these techniques, busy-looping and context switches, waste the hardware resource that can be otherwise utilized by the computation on other hardware threads. Thus, the inefficiency may not be an issue when a system has only I/O workloads; but it becomes detrimental when I/O workloads are consolidated with computation workloads. Efficiency can be improved by making these techniques less aggressive, e.g., enforcing a shorter timeout for polling. However, this sacrifices the effectiveness of these techniques and I/O performance.

We illustrate the inefficiency issue with polling and priority boosting using the experiments with two combinations of applications, Sockperf with `MatMul`, and `Redis` with PageRank. Sockperf and Redis are I/O-bound. MatMul and PageRank are CPU-bound. We run each combination on a 24-core server (48 hyperthreads) with each application running in a 48-vCPU VM. This results in 2 vCPUs on each hyperthread. The VMs are managed by KVM/Linux. Detailed server/VMs configurations and application descriptions can be found in §4.5.

Table 3.1 Existing Techniques Handling I/O Workloads Incur Frequent VCPU Switches and Massive Spinning, and Are Inefficient on SMT Processors. "VCPU Switches" Are Counts of Context Switches Between VCPUs Every Second in the Server. The Performance Improvements Are Relative to "Vanilla" KVM.

workloads	**KVM w/ enhanced HALT-Polling**			**vSMT-IO**		
	vCPU switches	spinning time	perf. imprv.	vCPU switches	spinning time	perf. imprv.
Sockperf	12.5K	40.1%	16.1%	3.3K	-	56.5%
MatMul			8.2%			57.4%
Redis	43.9K	27.5%	8.4%	15.1K	-	88.3%
PageRank			7.7%			123.1%

To illustrate the inefficiency issue on a well-tuned system with high efficiency, we have enhanced the HALT-Polling implementation in KVM. The enhancement makes HALT-Polling more effective, so as to further reduce context switches between vCPUs and make vCPUs more responsive to I/O events. Specifically, with the "vanilla" implementation, an idle vCPU is not allowed to perform HALT-Polling when there is another vCPU ready to run on the same hyperthread. The enhancement removes this restriction. It also increases the maximum timeout that is allowed in HALT-Polling. (HALT-Polling adjusts timeout value dynamically between 0 and a maximum value.) The enhancement improves the performance of the applications by 7.7% $\sim$ 16.1%.

As shown in Table 3.1, both application combinations incur frequent vCPU switches. For example, Redis and PageRank incur a vCPU switch about every 1 millisecond on each hyperthread. At the same time, a substantial portion of CPU time is spent by polling (e.g., 40.1% for Sockperf and `MatMul`). vCPU switches and such massive polling inevitably degrade performance, as we will show later.

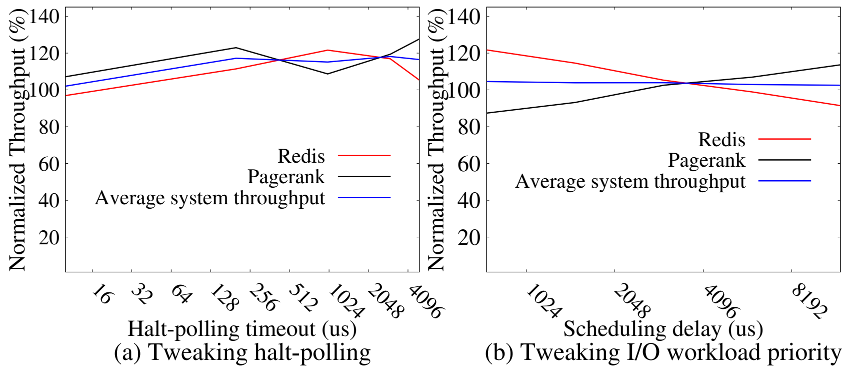


Figure 3.1 Tweaking existing techniques for scheduling I/O workload cannot substantially improve performance. (The throughputs are normalized to those with vanilla KVM.)

The performance advantage of the enhanced HALT-Polling is achieved by increasing polling to reduce costly vCPU switches. This demonstrates some potential to tweak existing designs. However, to improve performance significantly, major changes must be made. To illustrate this, Figure 3.1 shows how the performance of `Redis` and `PageRank` changes when tweaking the key parameters of polling and priority boosting. We first tweak the timeout used in HALT-Polling and vary it from 10 microseconds to 5 milliseconds. Figure 3.1(a) shows that increasing timeout only slightly improves performance when timeout value is small. However, the performance improvement of these two applications hits a plateau at about 10% after the timeout value reaches 200 microseconds.

Then, we adjust the scheduling delay parameter in Linux. The parameter controls the delay between a vCPU being woken up upon an I/O event and the vCPU preempting another vCPU. Thus, increasing the parameter essentially reduces the priority of I/O-bound vCPUs and reduces vCPU switches. As Figure 3.1(b) shows, the average performance barely changes; and increasing this parameter is basically sacrificing I/O performance for higher processor throughputs.

The aim of vSMT-IO is to substantially reduce the overhead caused by spinning and vCPU switches. The reduced overhead improves the performance of computation workloads. As shown in Table 3.1, reducing more than 2/3 of vCPU switches and eliminating spinning lead to significant performance improvement to `PageRank` (123.1% relative to vanilla KVM or 107.1% relative to enhanced KVM). More importantly, the performance improvement of computation workload is not at the cost of I/O performance. With vSMT-IO, the throughput of `Redis` is increased by 88.3% over vanilla KVM or 73.7% over enhanced KVM. The system I/O throughput is also increased by 75.1% over enhanced KVM.

3.2.2 Overhead of Polling and Context Switches

Existing techniques for improving I/O performance are inefficient on SMT processors, because context switches and polling waste the resource that can be otherwise utilized by the computation on other hardware threads. Targeting virtualized clouds, this subsection highlights the overhead of these operations with experiments and explains how such high overhead is incurred.

Table 3.2 VCPU Switches and HALT-Polling on A Hyperthread Slow Down the Computation on the Other Hyperthread.

Hyperthread 1	**Hyperthread 2**	**Relative performance**
-	`MatMul`	100%
vCPUs Switches	`MatMul`	32%
HALT-Polling	`MatMul`	73%

In the experiments, we run a `MatMul` thread on a hyperthread. Then, on the other hyperthread, we make two vCPUs switch back and forth or make a vCPU repeat the HALT-Polling loop. We check how the performance of `MatMul` is impacted by these operations.

The experiments show that vCPU switches slow down `MatMul` by about 70%, and HALT-Polling slows it down by about 30% (Table 3.2). While the slowdowns explain the inefficiency of polling and priority boosting techniques, we were surprised at these slowdowns. We expected the slowdown caused by vCPU switches to be around 50%, because there are two streams of instructions compete for CPU resource on the hyperthreads, and expected the slowdown caused by HALT-Polling to be minimal, because PAUSE instruction is designed to consume minimal resource.

We have diagnosed the slowdowns. vCPU switches cause large slowdowns mainly because the L1 data cache shared by both hyperthreads needs to be flushed during vCPU switches to address the *L1 Terminal Fault* problem [98; 99]. Other costly operations, including TLB flush [100], handling rescheduling IPIs [101], and the execution of scheduling algorithm, also contribute to the performance impact incurred by vCPU switches. The slowdown caused by HALT-Polling is larger than expected because the operations other than PAUSE are executed. HALT-Polling is implemented in the VMM. Thus, VM_EXIT is incurred when a vCPU enters HALT-Polling. VM_EXITs are costly operations [102]. During the polling, the instructions controlling the busy-loop are executed repeatedly. They are also more costly than PAUSE.

3.3 Basic Idea and Technical Issues

As Section 3.2 shows, polling and priority boosting incur high overhead on SMT processors; tweaking these techniques yields only marginal performance improvements. This requires that a new and efficient technique be developed to handle I/O workloads.

On a SMT processor, an efficient technique must consume minimal hardware resources. In a scheduling technique for improving I/O performance, two factors determine its hardware resource consumption. One is how to handle an I/O workload while it is waiting for the completion of an I/O operation. The other is how to quickly resume the execution of the I/O workload upon the completion of the expected I/O operation. Polling and priority boosting each concentrate on reducing the resource consumption of only one factor, but at the cost of high resource consumption in the other factor. Our solution aims to minimize the resource consumption of both factors.

Our solution leverages two features of SMT processors: 1) hardware-based blocking support, and 2) intense resource sharing between hardware threads. With these features, we implement a **Context Retention** mechanism for vCPUs. While a vCPU is waiting for the completion of I/O operations, it can "block" itself on a hardware thread, and release all its resources for other hardware threads to use, except for its hardware context. This minimizes the resource consumption required by waiting for the completion of I/O operations. With the hardware context, the vCPU can be quickly "unblocked" without context switches upon the completion of the I/O operations. This minimizes the resource consumption required to quickly resume the execution of I/O workloads. Table 3.3 summarizes the benefits of context retention from the perspectives of both I/O workloads and computation workloads.

Table 3.3 Summary of Benefit and Overhead of Context Retention.

	Benefit	**Overhead**
I/O	better responsiveness	timeslice charged for context retention
Computation	extra resources from reduced context switches and polling	resource underutilization

Though context retention consumes minimal hardware resources, it does incur some overhead, which are as summarized in Table 3.3 and must be reduced for better efficiency. From the perspective of computation workloads, because not all the hardware threads can be used for computation, the overhead is reflected by resource underutilization. Given that a x86 core has two hyperthreads, to avoid low utilization, one must be doing computation while the other does context retention. Even with this arrangement, full utilization may not be achieved.

From the perspective of I/O workloads, they are charged for vCPU usage while they retain contexts; so only short context retention periods are cheaper than descheduling and rescheduling vCPUs; but longer retention periods are not. This problem can be illustrated by the performance of I/O workload `Redis` in Figure 3.1(a). Increasing HALT-Polling timeout improves the performance of `Redis` when the timeout value is low. However, after the timeout exceeds 0.5 millisecond, further increasing the timeout degrades its performance. This is because, with a longer timeout, polling consumes more timeslice and reduces the timeslice available to the computation in `Redis`. Though polling is used in this experiment, if polling is replaced with context retention, the performance trend would be similar.

For the above overhead issues, a natural solution is to control the maximum length of context retention, such that extended context retention periods will not cause high overhead. However, this solution cannot deal with the overhead of the context retention periods that are relatively short. Reducing this overhead requires some enhancement in vCPU scheduling. For example, resource underutilization can be mitigated by scheduling a resource-demanding vCPU on a hyperthread when context retention is in progress on the other hyperthread; the vCPUs with much timeslice consumed by context retention can be compensated with extra timeslice.

In addition to the overhead issues, context retention also creates some challenge on the integration of symbiotic scheduling methods, which are needed for improving CPU performance. The key of symbiotic scheduling is to estimate how well a group of workloads can corun on a SMT core (i.e., symbiosis level) [103; 76; 77; 78; 79]. This is achieved by monitoring workload executions periodically. For instance, SOS (Sample, Optimize and Symbiosis) [75] samples workload executions periodically in sample phases to determine their symbiosis levels, and preferentially coschedules tasks with the highest symbiosis levels in symbiosis phases. Thus, existing symbiotic scheduling methods require that the resource demand of a workload change steadily during its execution. Due to context retention and burstiness of I/O operations [93], the resource demand of an I/O workload changes dramatically during its execution on a vCPU. Existing symbiotic scheduling methods cannot handle such workloads. This issue may be addressed by coscheduling I/O workloads with computation workloads, such that symbiosis levels can be lifted by overlapping context retention with resource-demanding computation. Existing symbiotic scheduling methods can still be used to handle computation workloads.

3.4 vSMT-IO Design

We implement our idea and address the technical issues in vSMT-IO. In this section, we present the overall architecture of vSMT-IO and its major components.

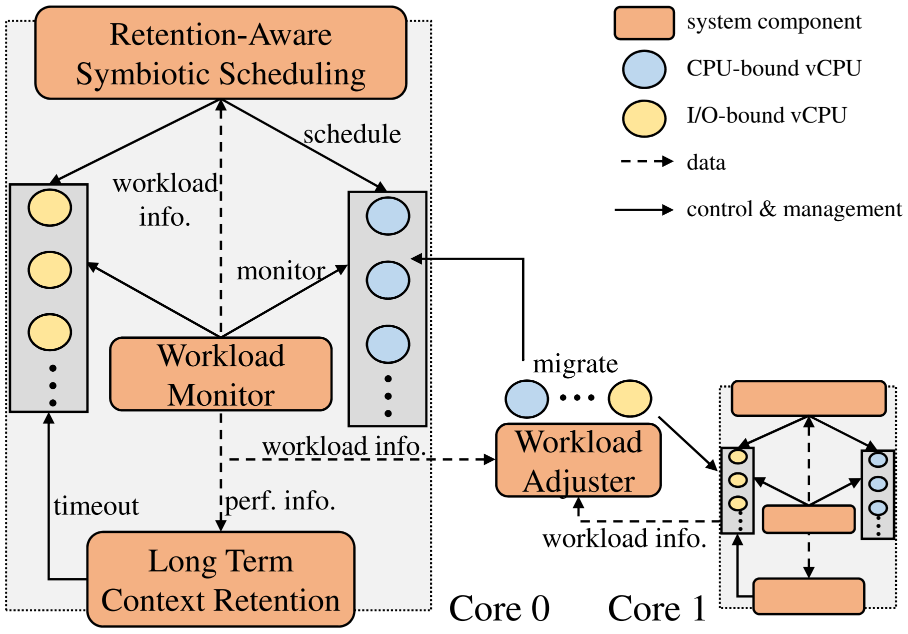


Figure 3.2 vSMT-IO Architecture. Key components are in orange.

3.4.1 Overview

Figure 3.2 shows the overall architecture of vSMT-IO. vSMT-IO incorporates four major components:

The **Long Term Context Retention (LTCR)** mechanism on each core implements context retention. To prevent extended context retention periods causing high overhead (resource underutilization and timeslice consumption), it enforces a context retention timeout, and dynamically adjusts the timeout value.

The **Retention Aware Symbiotic Scheduling (RASS)** algorithm is mainly to increase the symbiosis levels of the vCPUs running on the hypertheads in each core. To achieve this, RASS classifies vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs, and schedules CPU-bound vCPUs on a hyperthread and I/O-bound vCPUs on the other hyperthread. CPU-bound vCPUs run on both hyperthreads only when I/O-bound vCPUs are not ready to run. In this way, the resource-demanding computation on CPU-bound vCPUs can overlap to the greatest extent with the resource-conserving context retention periods on I/O-bound vCPUs. Increased symbiosis levels improve CPU performance and reduce the overhead of context retentions. At the same time, using a dedicated hyperthread for I/O-bound vCPUs allows them to use extra CPU time as a "compensation" for the timeslice charged in context retention periods, and further prevents them from being unfairly penalized.

The **Workload Monitor** on each core monitors vCPU executions. It characterizes the workloads on the vCPUs and measures performance. It provides workload information for RASS to classify and schedule vCPUs and for the workload adjuster introduced below to adjust the workloads between cores. It provides performance information for LTCR to adjust the timeout value.

The effectiveness of RASS relies on the heterogeneity of the workloads on each core, some being CPU-bound and some others being I/O-bound. The **Workload Adjuster** supplements RASS. It adjusts the workloads on each core to maintain their heterogeneity by migrating vCPUs between cores.

Algorithm 1 Context Retention Timeout Adjustment

```
1: $T_d$: desired timeout value; $T_e$: effective timeout value; $T_{init}$: initial timeout value; $P$: time period between two adjustments
2: $T_d \leftarrow T_{init}$
3: **while** true **do**
4:     $T_e \leftarrow T_d$, collect performance data for a time period of $P$
5:     **if** TESTTIMEOUT($T_d$ * 1.1) **then**
6:         $T_d \leftarrow T_d * 1.1$; continue
7:     **else**
8:         $T_e \leftarrow T_d$, collect performance data for a time period of $P$
9:     **end if**
10:    **if** TESTTIMEOUT($T_d$ * 0.9) **then**
11:        $T_d \leftarrow T_d * 0.9$; continue
12:    **end if**
13: **end while**
14: **function** TESTTIMEOUT($T$)
15:    $T_e \leftarrow T$, collect performance data for a time period of $P$
16:    $S_{cpu} \leftarrow$ average speed-up of CPU-bound vCPUs
17:    $S_{io} \leftarrow$ average speed-up of I/O bound vCPUs
18:    **if** $S_{cpu} > 1$ and $S_{io} > 1$ **then return** true; **end if**
19:    **return** false
20: **end function**
```

3.4.2 Long Term Context Retention (LTCR)

On x86 processors, we implement vCPU context retention with the vCPU Monitor/`mwait` support. Specifically, to wait for an I/O event, a vCPU calls a `mwait` instruction paired with a vCPU Monitor instruction that specifies a memory location in guest OS. The `mwait` instruction "blocks" the vCPU and keeps its context on the hyperthread. With the vCPU Monitor/`mwait` support, the `mwait` instruction ends automatically when the content at the memory location is updated or an interrupt is directed to the hyperthread. Since both I/O events and timeouts can be notified with interrupts, we choose to use interrupts to stop `mwait`. To prevent `mwait` from being terminated by memory writes prematurely, we set the memory location used in vCPU Monitor read-only.

The context retention timeout is to balance the cost and benefit of context rentention. Based on the summary in Table 3.3, for I/O workloads, lengthening a context retention is always a gain when it consumes less timeslice than descheduling and then rescheduling a vCPU. For computation workloads, context retention is rewarding when the amount of resource saved by reducing context switches and polling exceeds the amount of resource that cannot be utilized due to context retention. In the cases where one hyperthread does computation and the other hyperthread does context retention, context retention is always rewarding if it is not longer than the time spent on descheduling and then rescheduling a vCPU, based on the measurements shown in Table 3.2. Thus, context retention timeout can be set to be at least the time required by descheduling and rescheduling a vCPU. Then, longer timeouts can be tested.

LTCR uses algorithm 1 to adjust the context retention timeout periodically. The algorithm slightly increases or decreases the timeout value, checks whether performance is improved with the new value, and keeps the new value if it is. The algorithm uses the vCPU performance information collected by the workload monitor to determine whether performance is improved. Specifically, it uses IPC (instruction per cycle) to measure the performance of CPU-bound vCPUs, and uses the frequency of context retentions (i.e., number of context retentions per second) to measure the performance of I/O-bound vCPUs. Then, the algorithm calculates a speed-up for each vCPU. A speed-up value greater than 1 indicates that the performance of the vCPU has been improved with the new timeout value. It averages the speed-up values of CPU-bound vCPUs, and averages the speed-up values of I/O-bound vCPUs. The algorithm determines that the performance is improved and the new timeout value should be kept only if both average values are greater than 1.

3.4.3 Retention Aware Symbiotic Scheduling (RASS)

RASS schedules the vCPUs on each core with the main aim of maximizing the computation throughput of the core. This is achieved by increasing the symbiosis levels of the vCPUs running on the hypertheads. RASS combines two methods. One is *unbalanced scheduling* that maximizes the overlapping between resource-demanding computation and resource-conserving context retention periods (Section 3.4.3). The other is *symbiotic scheduling based-on cycle accounting* to select CPU-bound vCPUs with high symbiosis levels when both hardware threads need to run CPU-bound vCPUs (Section 3.4.3).

Unbalanced Scheduling Unbalanced scheduling classifies vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs, and schedules them on paired hyperthreads (See Figure 3.3). The classification is based on how much time each vCPU spends on context retention. Specifically, for each vCPU, a context retention rate is calculated and updated periodically. It is the ratio between the time spent on context retention in last time period and the period length. When a new period begins, the vCPUs are ranked based on their context retention rates. The vCPUs with higher context retention rates are considered to be I/O-bound, and the rest are CPU-bound.

When the hyperthread running I/O-bound vCPUs is idle, a CPU-bound vCPU is selected based on the symbiosis level (Section 3.4.3) and migrated to this hyperthread. This is to improve the utilization of CPU hardware to further increase CPU performance.

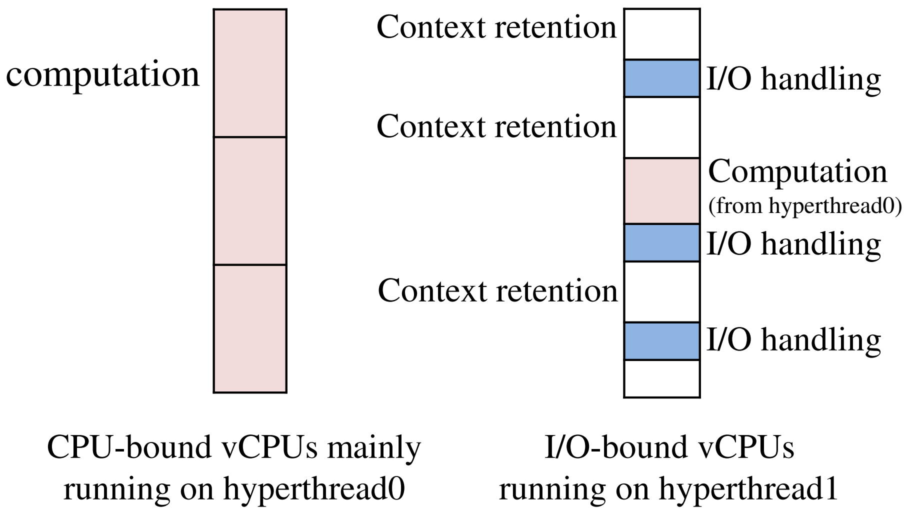


Figure 3.3 Computation and context retention are distributed to different hyperthreads with unbalanced scheduling.

The CPU-bound vCPU can only run with a priority lower than the I/O-bound vCPUs. It is preempted and migrated back when an I/O-bound vCPU becomes ready to run. This is to prevent the CPU-bound vCPU from blocking I/O-bound vCPUs and degrading I/O performance.

Unbalanced scheduling assumes that each vCPU has been attached with a weight, e.g., that used in Linux Completely Fair Scheduler (CFS). When classifying the vCPUs, it tries to balance the total weight of CPU-bound vCPUs and the total weight of I/O-bound vCPUs, and make them roughly equal. This is mainly to balance the load on the hyperthreads and reduce the migration of CPU-bound vCPUs.

The compensation to I/O-bound vCPUs for the timeslice consumed by context retentions can also be implemented by adjusting the weights of vCPUs. For example, the weights of the vCPUs can be increased based on their context retention rates. For the vCPUs that spend more time on context retentions than other vCPUs, their weights are increased by larger percentages. In this way, fewer vCPUs are classified as I/O-bound, and share the same hyperthread. However, we found that this adjustment is not necessary in most cases. The main reason is that I/O-bound vCPUs usually have low CPU utilization. Thus, even with context retention, some I/O-bound vCPUs still cannot fully consume their timeslice. Other I/O-bound vCPUs that need more timeslice acquire automatically extra timeslice as compensation. This is because the scheduler is work-conserving, and I/O-bound vCPUs have higher priority than CPU-bound vCPUs on the hyperthread and are supplied with extra timeslice first.

Symbiotic Scheduling Based on Cycle Accounting When both hyperthreads need to run CPU-bound vCPUs, the symbiosis levels between vCPUs must be considered. RASS determines the symbiosis levels using the cycle accounting technique [104; 105; 106; 107]. It is a symbiotic scheduling technique for threads. We only adapt its method that estimates the symbiosis levels between threads and use it on vCPUs.

We select this technique because of its high practicality. To estimate the symbiosis levels between threads, it samples and characterizes each individual thread, and inputs the characterization into an interference estimation model. Compared to SOS (Sample, Optimize and Symbiosis), which samples the execution of possible thread combinations [75], the cycle accounting technique has a much lower complexity ($O(n)$ vs. $O(n^2)$) and thus higher practicality.

The cycle accounting technique uses three parameters, which are the components of the CPI (cycler per instruction), to characterize a thread. The **b**ase component (**B**) is the number of cycles used to finish an instruction when all the required hardware resource and data are locally available; the **m**iss component (**M**) is the number of cycles used to handle misses (e.g., cache misses and TLB misses); the **w**aiting component (**W**) is the number of cycles waiting for hardware resource to become available. The CPI value is roughly the sum of B, M, and W.

When the parameters of a thread are being measured, the cycle accounting technique requires that the thread run alone on the core without any computation on the other hyperthread so as to eliminate interference. This incurs non-trivial overhead. To reduce this overhead, we take advantage of context retentions, and measure the parameters of a CPU-bound vCPU when it is running on a hyperthread and context retention is in progress in the other hyperthread. We obtain the base component, the miss component, and the CPI of the vCPU using hardware counters, and calculate the waiting component from this.

3.4.4 Workload Adjuster

The effectiveness of RASS relies on the heterogeneity of the workloads on each core, some being CPU-bound and some others being I/O-bound. Its performance advantage may diminish when workloads become homogeneous due to factors, such as load balancing and phase changes in workloads. The workload adjuster is designed to maintain the workload heterogeneity on each core.

The workload adjuster measures workload heterogeneity and characterizes the overall workload type by calculating the standard deviation and the average value of vCPU context retention rates. If a group of vCPUs have a small deviation value, their workloads are generally homogeneous; if the average context retention rate of a group of vCPUs is very high, these vCPUs are likely to be I/O-bound; if the average rate is very low, the vCPUs are likely to be CPU-bound. The workload adjuster calculates these values for each core, and updates them periodically to detect the need for workload adjustment. When the standard deviation drops below a pre-set threshold, workload adjustment starts.

To adjust the workloads, the adjuster finds the core with the smallest deviation. Then, based on the average context retention rate of the core (e.g., a very small average value of CPU-bound vCPUs), the adjuster searches for another core, which is dominated by the other type of vCPUs (e.g., I/O-bound vCPUs). The search is done by examining the average context contention rates of other cores. The desired core is the one with the average context contention rate that differs from the former average rate by the largest degree (e.g., a very large average value of I/O-bound vCPUs). After a such core is found, the adjuster ranks the vCPUs based on their context retention rates on each of these two cores, selects the vCPU ranked in the middle on each core, and swaps the two vCPUs.

3.5 Implementation Details

We have implemented a prototype of vSMT-IO based on Linux/KVM. We added/-modified about 1300 lines of source code mainly in KVM kernel modules and Linux CFS [2]. The workload monitor and the long-term context retention (LTCR) components are mainly implemented in a KVM kernel module by changing *kvm_main.c*. In LTCR, the context retention mechanism needs to be implemented in guest OS to minimize overhead. Though it can be implemented as an idle driver kernel module [108], we choose to directly change the idle loop in *idle.c* to simplify the implementation. Context retention is implemented with a loop, which repeatedly calls vCPU Monitor, `mwait`, and the *need_sched()* function of Linux kernel. It is inserted at the beginning of each iteration of the idle loop. Implementing context retention with a loop is to prevent it from being terminated prematurely by irrelevant interrupts. The loop terminates when a thread becomes “ready” on the vCPU (fulfilled with the *need_sched()* call). Thus, context retention can finish upon the expected I/O event. The loop also ends if a timer interrupt “marking” the timeout of the context retention is received by the vCPU. To differentiate this interrupt from regular timer interrupts, we change the two unused bits in the VM execution control register, and use them as a timeout flag.

Retention aware symbiotic scheduling and workload adjuster are implemented based on Linux CFS in *fair.c* and *core.c*. Thus, the original scheduling and load balancing policies implemented in CFS are followed in most cases, e.g., when deciding which I/O-bound vCPU is the next to run on a hyperthread. However, when deciding which CPU-bound vCPU is the next to run, the symbiotic scheduling policy in RASS and the fairness based scheduling policy in CFS have different objectives, and thus may decide to select different vCPUs. To coordinate these different objectives, our implementation let Linux CFS select a few vCPUs based on its policies. Then, among these vCPUs, RASS selects a vCPU based on symbiosis.

3.6 Performance Evaluation

With the prototype implementation, we have evaluated vSMT-IO extensively with a diverse set of workloads. The objectives of the evaluation are four-fold: 1) to show that vSMT-IO can improve I/O performance with high efficiency and benefit both I/O workload and computation workload, 2) to verify the effectiveness of the major techniques used in vSMT-IO, 3) to understand the performance advantages of vSMT-IO across

[2]Source code can be found at https://github.com/vSMT-IO/vSMT-IO.

diverse workload mixtures and different scenarios, and 4) to evaluate the overhead of vSMT-IO.

3.6.1 Experiment Settings

Our evaluation was done on a DELL^{TM} PowerEdge^{TM} R430 server with two 2.60GHz Intel Xeon E5-2690 processors (two NUMA zones), 64GB of DRAM, a 1TB HDD, and an Intel I350 Gigabit NIC. Each processor has 12 physical cores, and each physical core has two hyperthreads. With KVM, we built four VMs, each with 24 vCPUs and 16GB memory. Both the host OS and guest OS are Ubuntu Linux 18.04 with kernel updated to 5.3.1. We test vSMT-IO with a large and diverse set of workloads generated by typical applications from different domains, as summarized in Table 3.4. In the experiments, each VM encapsulates one workload.

We test vSMT-IO under two settings. Under the first setting, we launch two VMs; thus each vCPU has a dedicated hyperthread. We compare vSMT-IO against three competing solutions: 1) `Blocking`, which immediately deschedules the vCPUs waiting for I/O events, and is implemented by disabling HALT-Polling in KVM; 2) Polling, which is implemented by booting guest OS with parameter *idle=poll* configured [109] (timeout is not enforced for best I/O performance); and 3) HaltPoll implemented in KVM, which combines polling and priority boosting techniques.

Under the second setting, we launch four VMs; thus, each hyperthread is time-shared by two vCPUs. Without a timeout, Polling is not a choice for improving I/O performance under this setting. Thus, we compare vSMT-IO against 1) vanilla KVM, which uses `priority boosting` to improve I/O performance, because HaltPoll implemented

Table 3.4 Benchmark Applications Used to Test vSMT-IO.

App.	**Workload Description**
`Redis`	Serve requests (randomly chosen keys, 50% SET, 50% GET) [110].
`HDFS`	Read 10GB data sequentially with `HDFS TestDFSIO` [51].
`Apache Hadoop`	`TeraSort` with `Apache Hadoop` [51].
`HBase`	Read and update records sequentially with YCSB [54].
`MySQL`	OLTP workload generated by SysBench for `MySQL` [111].
`Nginx`	Serve web requests generated by `ApacheBench` [112].
`ClamAV`	Virus scan a large file set with clamscan [49].
`RocksDB`	Serve requests (randomly chosen keys, 50% SET, 50% GET) [113].
`PgSql`	TPC-B-like workload generated by PgBench [114].
`Spark`	`PageRank` and `Kmeans` algorithms in `Spark` [115].
`DBT1`	TPC-W-like workload [116].
`XGBoost`	Four AI algorithms included in `XGBoost` [117] system.
`MatMul`	Multiply two 8000x8000 matrices of integers.
Sockperf	TCP ping-pong test with Sockperf [118].

in vanilla KVM is inactive under this setting, and 2) HaltPoll enhanced to support time-sharing (described in Section 3.2.1).

We measure the throughputs of the workloads. We also collect response times if the workloads report them. The performance measurements may vary significantly across different workloads. When we present them in figures, for clarity, we normalize them against those of `Blocking` under the first setting and `priority boosting` (i.e., vanilla KVM) under the second setting.

3.6.2 One vCPU on Each Hyperthread

Under the first setting, I/O workloads can achieve the best performance with Polling. We want to compare the effectivenss of vSMT-IO on improving I/O performance against that of Polling by comparing the performance of I/O workloads managed with these two solutions. Without a timeout, Polling incurs high overhead on SMT processors, and degrades the performance of other workloads on the processors. `Blocking` and HaltPoll are more efficient solutions than Polling under this setting. We want to compare the efficiency of vSMT-IO against that of `Blocking` and HaltPoll by comparing the performance of computation workloads when they are collocated with I/O workloads managed with these three solutions.

With the above objectives, we launch two VMs. We run `MatMul` in one VM, which is computation-intensive, and run an I/O-intensive benchmark in the other VM. Figure 3.4 shows the normalized throughputs of `MatMul` and eight I/O-intensive benchmarks selected to co-run with `MatMul`. Note that the performance with `Blocking` is shown with the flat line at 100%.

With vSMT-IO, the I/O-intensive benchmarks achieve similar performance as they do with Polling. The largest difference is with `DBT1`, 4.1%. This is because `DBT1` incurs a large number of random accesses to the HDD, which have long latencies exceeding the timeout value used in LTCR. On average, the I/O intensive benchmarks are only 2.3% slower with vSMT-IO. This shows that vSMT-IO is highly effective on improving I/O performance.

The high effectiveness of vSMT-IO is achieved with high efficiency. This is reflected by `MatMul` achieving higher performance with vSMT-IO consistently in all the experiments than it with the other three solutions. On average, with vSMT-IO the performance of `MatMul` is 37.9%, 14.5%, and 27.6% higher than it with Polling, `Blocking`, and HaltPoll, respectively.

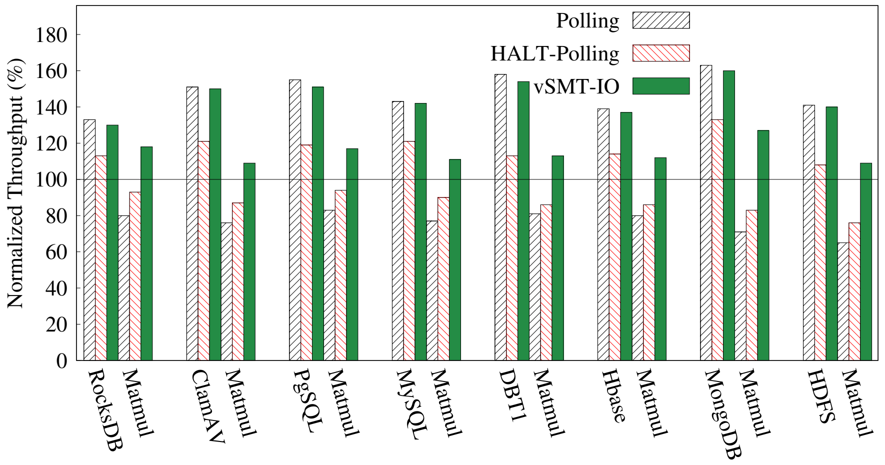


Figure 3.4 Throughputs of `MatMul` and eight I/O-intensive benchmarks when `MatMul` is collocated with each of the benchmarks in two VMs. Each vCPU runs on a dedicated hyperthread. Throughputs are normalized to those of `Blocking`.

3.6.3 Multiple vCPUs Time-Sharing a Hyperthread

With multiple vCPUs on each hyperthread, context switches are usually incurred when improving I/O performance. It becomes more difficult for I/O-improving solutions to maintain high efficiency. We want to know to what extent the effectiveness and efficiency of vSMT-IO can be maintained. At the same time, vSMT-IO can be fully exercised under this setting. We want to verify the effectiveness of the major techniques in vSMT-IO.

In the experiments, we launch four VMs. On two of the VMs, we run two instances of the same benchmark, which is computation-intensive, e.g., `Nginx`, or AI algorithms in `XGBoost`. On the other two VMs, we run two instances of another benchmark, which is I/O-intensive, e.g., web server, or file server.

Figure 3.5 shows the normalized throughputs for eight benchmark pairs. In each pair, the first benchmark is I/O intensive, and the second benchmark is computation intensive. The enhanced HaltPoll can effectively improve the throughputs of I/O-intensive benchmarks, because polling can "absorb" some context switches caused by I/O operations. Compared to vanilla KVM, the throughputs of I/O intensive benchmarks are increased by 36.9% on average. However, polling consumes CPU resources and may degrade the performance of other workloads (e.g., `Nginx` and Regression). Because the length of polling is carefully controlled in HaltPoll, on average the throughputs of computation-intensive benchmarks are similar to those with vanilla KVM.

Compared to enhanced HaltPoll, vSMT-IO can more effectively improve the throughputs of I/O-intensive benchmarks. On average, their throughputs are 29.5% higher than those with enhanced HaltPoll. More importantly, this is achieved by improving the throughputs of computation-intensive workloads at the same time. On average, the throughputs of computation-intensive workloads with vSMT-IO are 22.8% and 18.4% higher than those with enhanced HaltPoll and vanilla KVM, respectively.

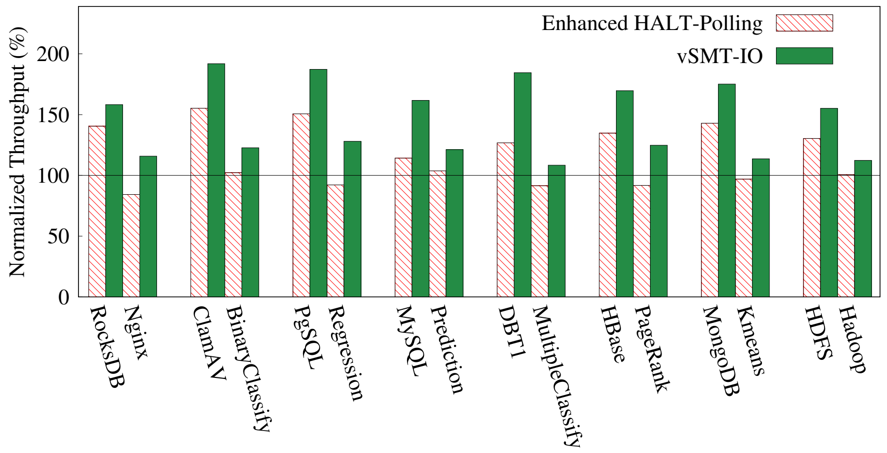


Figure 3.5 Throughputs of eight pairs of benchmarks. Each benchmark has two instances running on two VMs. Each hyperthread is time-shared by 2 vCPUs. Throughputs are normalized to those with vanilla KVM. Benchmarks BinaryClassify, MultipleClassify, Regression and Prediction are AI algorithms in `XGBoost`.

The results in Figure 3.5 confirm that vSMT-IO can maintain its effectiveness and efficiency when each hyperthread is time-shared by vCPUs. To further investigate how the throughputs are improved with vSMT-IO, we collect the frequencies of vCPU switches (shown in Table 3.5) and profile the workload on the hyperthreads for I/O-bound vCPUs (results shown in Table 3.6).

Table 3.5 Number of VCPU Switches is Substantially Reduced with vSMT-IO for the Eight Benchmark Pairs.

Benchmark Pairs	**Number of vCPU Switches Per Second**		
	Vallina KVM	**Enhanced HaltPoll**	**vSMT-IO**
(RocksDB,Nginx)	29.3k	15.2k	1.9k
(ClamAV,BinaryClassify)	11.8k	8.7k	3.2k
(PgSql,Regression)	9.5k	8.0k	2.8k
(MySQL,Prediction)	11.5k	9.3k	4.5k
(DBT1,MultipleClassify)	61.3k	29.5k	3.9k
(HBase,PageRank)	23.4k	12.3k	3.9k
(MongoDB,Kmeans)	33.3k	20.8k	9.3k
(HDFS,Apache Hadoop)	34.0k	30.6k	1.7k

The effectiveness of vSMT-IO on improving I/O performance relies on context retentions holding vCPU contexts on hyperthreads (the LTCR component). It is reflected by reduced context switches. As shown in Table 3.5, vSMT-IO can reduce vCPU switches significantly by up to 95% (80% on average). As a comparison, enhanced HaltPoll can only reduce vCPU switches by at most 51% (32% on average). This explains the superiority of vSMT-IO over HaltPoll.

The high efficiency of vSMT-IO comes partially from its capability to reduce vCPU switches. It also comes from LTCR and RASS controlling the overhead incurred by context retentions. While the effectiveness of RASS on controlling the overhead

Table 3.6 Time (Percentage) Spent By Context Retentions, I/O-bound VCPU, and CPU-bound VCPU on the Hyperthreads for I/O-bound VCPUs.

Benchmark Pairs	**Context Retentions**	**I/O Workload**	**Computation Workload**
(`RocksDB`,`Nginx`)	28.1%	34.3%	37.6%
(`ClamAV`,BinaryClassify)	39.8%	31.6%	28.6%
(`PgSql`,Regression)	42.3%	19.2%	38.5%
(`MySQL`,Prediction)	30.0%	33.5%	36.5%
(`DBT1`,MultipleClassify)	32.7%	54.4%	12.9%
(`HBase`,`PageRank`)	53.9%	31.9%	14.2%
(`MongoDB`,`Kmeans`)	34.4%	45.3%	20.3%
(`HDFS`,`Apache Hadoop`)	33.0%	45.2%	21.8%

is self-evident, the effectiveness of LTCR can be confirmed with the results shown in Table 3.6. LTCR limits the context retention lengths to prevent high overhead. As a result, on the hyperthreads for I/O-bound vCPUs, for most benchmark pairs, the time spent on context retentions is less than 40%. With context retention lengths well controlled, more than 20% of the CPU time on these hyperthreads can be used by CPU-bound vCPUs to improve CPU throughput.

To understand how the two major techniques in vSMT-IO, xWait and Unbalancer, improve performance, we enable these techniques separately, and show the performance of two pairs of benchmarks, `HBase` with `PageRank`, and `MongoDB` with `Kmeans`, in Figure 3.6. xResolver is enabled along with Unbalancer, because it is a supplement to Unbalancer. Figure 3.6 shows that the performance improvements of I/O-intensive workloads are mainly from the xWait technique; and the performance improvements of computation-intensive workloads are mainly from the Unbalancer technique. When xWait is enabled, the throughputs of I/O-intensive workloads, `HBase` and `MongoDB`, are significantly increased by 41.1% and 44.7%, respectively. However, it barely increases the

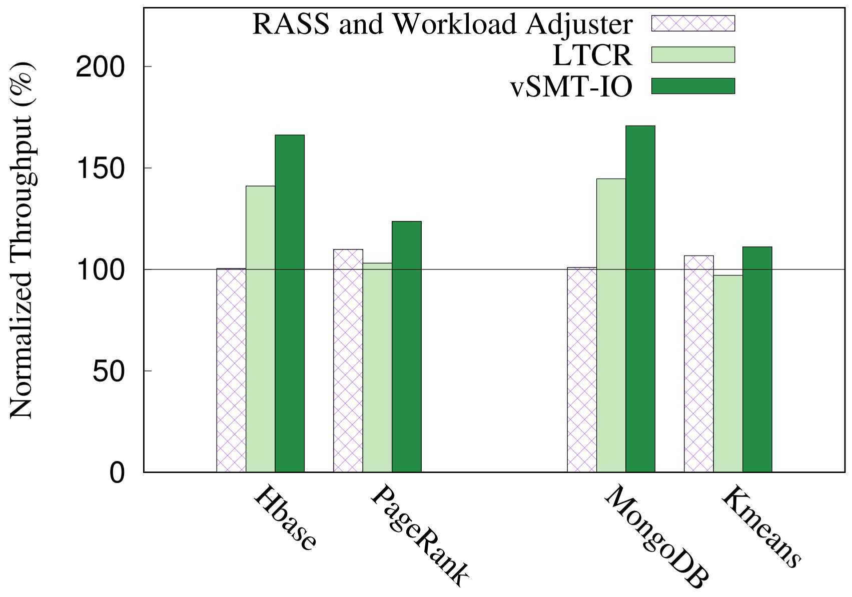


Figure 3.6 Normalized throughputs (relative to those achieved with vanilla KVM) of two pairs of benchmarks when xWait and Unbalancer are enabled separately.

throughputs of `PageRank` and `Kmeans`. Further enabling Unbalancer (with xResolver) can effectively improve the throughputs of all the workloads.

Some benchmarks report response times. Figure 3.7 compares how their response times are reduced with vSMT-IO and HaltPoll. Relative to vanilla KVM, HaltPoll reduces the response times by 28.7% on average. vSMT-IO can reduce the response times by larger percentages (50.8% on average). To investigate how vSMT-IO reduces response times, we monitor the state changes of the vCPUs during the executions of these benchmarks, collect the time spent by vCPUs at the following states: 1) ***Running***, including context retention, on a hyperthread, 2) ***Ready*** and waiting to be scheduled, 2) ***Waiting*** for an event. In Table 3.7, for each benchmark, we show the time (in milliseconds) spent in these states for serving a request.

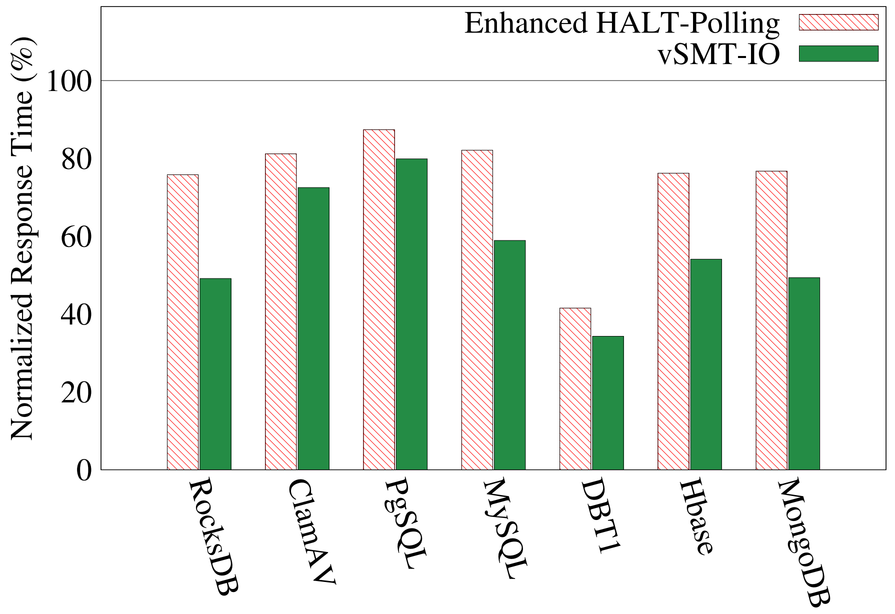


Figure 3.7 Response times of `RocksDB`, `ClamAV`, `PgSql`, `MySQL`, `DBT1`, `HBase`, and `MongoDB` normalized to those with vanilla KVM (shown with the horizontal line at 100%).

Table 3.7 Time Spent By VCPUs in Three States When Processing a Request with Vanilla KVM, Enhanced HaltPoll, and vSMT-IO.

Benchmark	**Vallina KVM**			**Enhanced HaltPoll**			**vSMT-IO**		
	Run	**Ready**	**Wait**	**Run**	**Ready**	**Wait**	**Run**	**Ready**	**Wait**
`RocksDB`	116.2	132.6	378.1	131.7	88.0	305.4	129.8	69.0	237.2
`ClamAV`	15.2	45.7	10.9	12.9	29.7	10.5	11.0	21.1	9.5
`PgSql`	14.7	37.6	10.1	12.3	27.1	9.4	13.5	19.7	8.7
`MySQL`	111.4	319.7	88.9	90.7	167.9	89.5	87.5	136.7	80.6
`DBT1`	346.2	1831.4	1390.2	361.6	842.9	1035.8	306.9	643.2	641.6
`HBase`	266.2	655.0	901.8	237.8	323.3	795.9	241.6	315.0	654.3
`MongoDB`	376.1	528.6	1444.1	365.3	345.2	1276.6	351.7	256.0	897.9

The response times are reduced with vSMT-IO mainly because vCPUs spend less time on waiting to be scheduled or for events. As shown in Table 3.7, vSMT-IO can significantly reduce the time in the *Ready* state (53.6% on average). This is because context retention reduces context switches between vCPUs, and thus reduces the scheduling delay associated with the switches. We have noticed that the time in the *Waiting* state is substantially reduced for some benchmarks (e.g., `DBT1`). This is because finishing an I/O operation sometimes need the collaboration of multiple vCPUs in the VM. For example, after a vCPU sends out an I/O request and becomes idle, another vCPU may receive the response and must notify the former vCPU by sending it an inter-processor interrupt (IPI). In this case, reducing the *Ready* time of the latter vCPU (i.e., scheduling it earlier) can also reduce the *Waiting* time of the former vCPU.

3.6.4 Applicability and Overhead

vSMT-IO targets heterogeneous workloads with intensive I/O operations and heavy computation. We want to know how well vSMT-IO performs for the workloads with different heterogeneity. This subsection tests the performance and overhead of vSMT-IO for different workload mixes. We still use 4 VMs to run 4 instances of 2 applications in the experiments. But we change VM sizes (i.e., the number of vCPUs in a VM) to change the workload mix. For example, to make the workload more I/O-intensive, we increase the sizes of the 2 VMs running I/O-intensive benchmarks and reduce the sizes of the VMs running computation-intensive benchmarks. The total number of vCPUs of the 4 VMs is kept fixed (96 vCPUs).

Figure 3.8 shows the normalized throughputs of two benchmark paris, `HBase` with `PageRank`, and `MongoDB` with `Kmeans`, when the VM sizes for I/O-intensive benchmarks and computation-intensive benchmarks are changed from (12,36) to (36,12). (The ratios of the vCPUs running these benchmarks vary from 24:72 to 72:24.) Figure 3.9 shows

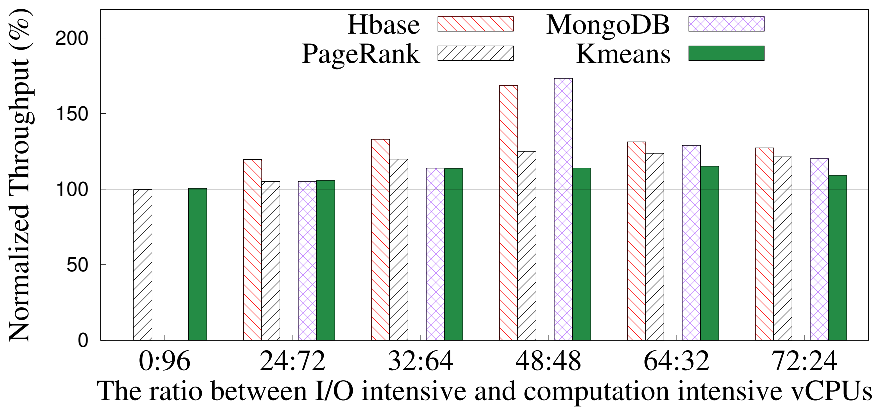


Figure 3.8 Normalized throughputs of vSMT-IO under different workload mixes. Throughputs are normalized to those with vanilla KVM.

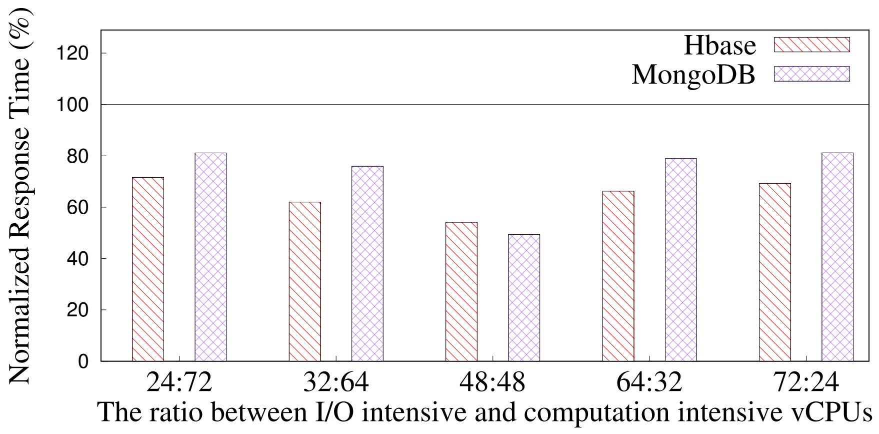


Figure 3.9 Normalized response times of vSMT-IO under different workload mixes. Response times are normalized to those with vanilla KVM.

the response times of `HBase` and `MongoDB` in these experiments. Though vSMT-IO can improve performance for all these workload mixes, it improves performance by the largest percentages when the number of vCPUs running I/O-intensive benchmarks is the same as the number of vCPUs running computation-intensive vCPUs.

We also run `PageRank` and `Kmeans` in two VMs with 48 vCPUs each, and show the normalized throughputs (labeled with "0:96") in Figure 3.8. Because both benchmarks are computation intensive, there is no space for vSMT-IO to improve performance. The performance difference between vSMT-IO and vanilla KVM is unnoticeable (less than 2%). This shows that the overhead of vSMT-IO is very low.

We have also evaluated the performance of vSMT-IO with 8 VMs (192 vCPUs). We find that vSMT-IO consistently shows better performance than vanilla KVM and enhanced HaltPoll, for heterogeneous workloads; but the performance improvement is similar to that with 4 VMs. The performance advantage of vSMT-IO is more determined by the mix of workloads than the number of VMs on each server.

3.7 Related Work

Improving I/O performance in virtualized systems. I/O performance problems in virtualized systems have been intensively studied; and various solutions have been proposed, including shortening time slices [119; 32; 120; 121], task-aware priority boosting [33; 61; 17; 36; 62; 63; 64; 65; 66; 67; 68; 69; 70], and task consolidation [122; 123; 124; 125; 87]. These solutions are not designed for SMT processors, and are orthogonal to our work. Shortening time slices of vCPUs can reduce the latency of I/O workloads in virtualized systems. However, it may incur significant performance degradation caused by context switches. Task-aware priority boosting improves I/O performance in virtualized systems by prioritizing I/O-intensive workloads. For instance, xBalloon [17] maintains the high priority of I/O-intensive workloads by reserving CPU resource for them. However, this may hurt the performance of computation-intensive workloads. vMigrater [87] prioritizes I/O-intensive workloads by migrating them away from to-be-descheduled vCPUs to other vCPUs, such that they can keep running and generating I/O requests. However, it is designed for VMs with multiple vCPUs, and may incur high workload migration cost. Task consolidation solutions can improve I/O performance by reducing the descheduling and rescheduling of vCPUs. They consolidate workloads onto fewer vCPUs if the workloads are I/O-intensive, such that these vCPUs can be kept active with relatively low cost. These solutions may also incur high cost due to frequent workload migrations. Polling is used in these solutions to keep vCPUs active. This is inefficient on SMT processors and can be improved by replacing polling with context retention.

Symbiotic scheduling aims to maximize the throughput of SMT processors by selecting the tasks with complementary resource demands and coscheduling them on the same SMT core [75; 76; 77; 78; 79; 80]. For instance, SOS (Sample, Optimize, Symbiosis) and its variants [75; 103; 76; 77; 78; 79; 80] sample task executions when they are coscheduled onto the same core, and preferentially coschedule those with small slowdowns. These solutions only target processor throughput, and cannot be used to improve the performance of I/O-intensive workloads.

Other scheduling solutions for SMT processors. Instead of maximizing processor throughput, some scheduling solutions aim to secure resources for individual tasks on SMT processors to ensure their decent performance [81; 103; 126; 127]. For instance, ELFEN [81] aims to ensure the high performance of latency-critical tasks when they are collocated with batch tasks on SMT processors. It puts a latency-critical task and batch tasks on different hardware threads in the same core, and “blocks” batch tasks when the latency-ciritical task is making progress. The efficiency is low with this solution, because each core has only one active hardware thread at any moment, and resource is underutilized. Tasks on the same SMT processor may not share the resources in a fair way. Various solutions have been proposed to enforce fairness among the tasks in a SMT-enabled system [128; 129; 130]. For instance, progress-aware scheduler [128] periodically estimates the progress of tasks, and prioritizes the tasks with relatively slow progress. vSMT-IO is orthogonal to these solutions. It increases efficiency to improve both CPU performance and I/O performance.

3.8 Conclusion and Future Work

Despite the prevalence of SMT processors, the problems with how to improve I/O performance and efficiency on SMT processors are surprisingly under-studied. Existing techniques used in CPU schedulers to improve I/O performance are seriously inefficient on SMT processors, making it difficult to achieve high CPU throughput and high I/O throughput. Leveraging the hardware feature of SMT processors, the paper designs vSMT-IO as an effective solution. The key technique in vSMT-IO is context retention. vSMT-IO targets virtualized clouds and x86 systems and addresses a few challenges in implementing context retention in real systems. Extensive experiments confirm its effectiveness.

NUMA systems have become ubiquitous. Though our evaluation demonstrates that vSMT-IO achieves better performance than competing solutions, the designs in vSMT-IO have not been optimized for NUMA systems. As future work, we want to make vSMT-IO "NUMA-aware" to further improve its performance. For example, the workload adjuster can be enhanced by adjusting workloads within each NUMA node before it migrates vCPUs across NUMA nodes.

CHAPTER 4

JUPITER: EFFECTIVE AND QOS-AWARE COSCHEDULING FOR MULTI-THREADED WORKLOADS IN MULTI-TENANT CLOUDS

4.1 Introduction

It has been widely noticed and intensively studied that the performance of multi-threaded applications is highly vulnerable to the time-sharing of CPUs [11; 131; 132; 133; 134; 135; 136; 137]. The vulnerability incurs a few performance issues compromising the Qualify of Service (QoS) for multi-threaded applications, such as the bad performance, the unfair performance penalty, low CPU utilization, and the sensitivity of the performance to other applications in the system.

The emergence of cloud computing and virtualization, where CPU cores are time-shared by multiple virtual machines (VMs), makes it more pressing to solve these problems for multi-threaded applications. First, an increasing number of applications become multi-threaded and run in clouds for high performance and low cost. However, the low performance caused by the performance vulnerability may make running multi-threaded applications in clouds cost-ineffective. Second, the abstraction of VMs gives an illusion of dedicated computers to users, making the users have high expectations for performance and performance isolation. However, for multi-threaded workloads, such expectation can hardly be met, because their performance is more sensitive to the workloads in other VMs collocated in the same server. This leads to frustrating user experiences.

To mitigate these QoS issues of multi-threaded applications in multi-tenant clouds, we propose techniques to fundamentally enhance coscheduling. Coscheduling [11] is a widely-used approach for reducing the performance vulnerability of multi-threaded applications. Various coscheduling schemes [138; 139; 140; 141] have been designed for OSs and virtual machine monitors (VMMs). Coscheduling aims to reduce the execution delay of multi-threaded applications at their synchronization/communication points, which is the main cause of their performance vulnerability. The main idea is to maximize the co-running of collaborating threads; i.e., when a thread is scheduled to run, its collaborating threads should be scheduled as quickly as possible so as to run in parallel with the thread. To maximize the co-running of collaborating threads, most co-scheduling schemes temporarily prioritize these threads, such that they can preempt the execution of other threads to get the cores to co-run and are less likely to be preempted by other threads during the co-running.

Unfortunately, the effectiveness of existing coscheduling approaches is seriously limited when used in multi-tenant clouds due to the following three reasons. First, when a system has two or more multi-threaded applications, the effectiveness of the existing coscheduling approaches is limited, because there are conflicting demands for prioritizing different applications on the same set of hardware. Existing approaches focus mainly on one multi-threaded application and lack a mechanism to resolve the conflicts from multiple multi-threaded applications. Even worse, existing coscheduling approaches indiscriminately prioritize the threads to be co-scheduled in each application, significantly increasing the likelihood of conflicts.

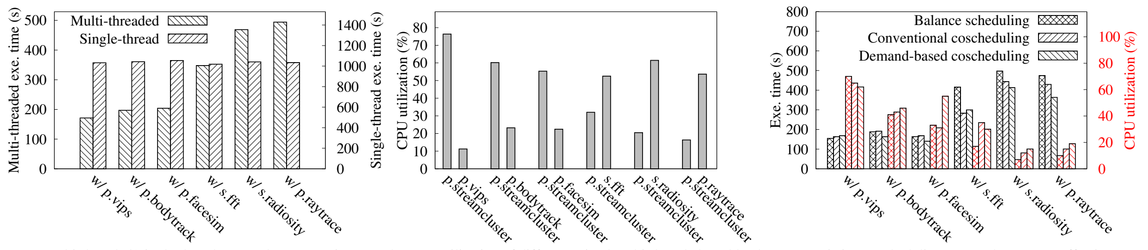
Second, existing coscheduling schemes cannot address well the trade-off between improving the effectiveness and reducing the notorious adverse effects of coscheduling. They may unnecessarily sacrifice effectiveness when trying to reduce the adverse effects. For example, relaxed coscheduling [138] reduces CPU fragmentation by coscheduling fewer threads; this is at the cost of lower application performance, as our experiments show in §4.3.

Third, the performance vulnerability problem becomes even more pronounced when applications having dynamic changing workloads (e.g., the workload variation caused by changing parallelism). The scheduler usually provisions time slice periodically at a fixed rate, and does not allow unused time slice in the periods with the light workload to be accumulated and used later when the workload is heavy. Thus, such workload changes are penalized. Existing coscheduling approaches lack a mechanism to deal with such a performance penalty.

In this work, we address the above issues of coscheduling with three novel ideas. First, prioritize threads in an asymmetric way. The observation is as follows. Existing coscheduling approaches prioritize collaborating threads indiscriminately. This unnecessarily increases prioritized executions, and thus increases their conflicts and adverse effects. However, it is not necessary to boost the priority of all the collaborating threads to the same high level. For example, some threads can quickly produce the data required by other threads; even if they are scheduled late, they may still be able to produce the data before other threads need it; thus, there is no need to boost the priority of these threads. Second, prioritize threads less aggressively to further reduce the adverse effects of coscheduling without reducing its effectiveness. Coscheduling can be done in a less aggressive way as long as the application performance is not reduced. Third, effectively combine coscheduling with the leaky bucket technique [142] to deal with the vulnerability caused by the dynamic changing workload.

Targeting virtualized multi-tenant clouds, we implement our idea in JUPITER, an effective and QoS-aware coscheduling approach for virtual CPUs (vCPU). The general ideas and techniques developed in the paper can be universally applied to other types of virtualized environments (e.g., containers) and non-virtualized environments in conventional operating systems. We choose to implement and test our ideas in VMM for two reasons: 1) in multi-tenant clouds, where virtual machines are dominantly used, performance isolation and QoS issues are more critical than in other environments; 2) due to the semantic gap between the VMM and the guest systems, the VMM has less information about the workloads in guest systems, making the problems more challenging with VMs. If our ideas and techniques can be implemented at the VMM level, and are tested to be effective, they may also be implemented in other environments and work effectively.

This paper makes the following major contributions. **First**, the paper identifies and addresses three critical issues with coscheduling to substantially improve the QoS for multi-threaded applications in clouds while minimizing the adverse effects caused by co-scheduling. **Second**, it implemented JUPITER in Linux/KVM [143]. JUPITER runs in the VMM layer, and thus is transparent to applications and VMs. **Third**, it evaluated JUPITER on a diverse set of 7 programs, including two server programs `MongoDB` [24] and `PgSql` [114], four popular AI programs in `XGBoost` [144] library, a widely used TPC-W [116] benchmark, a parallel compression utility PBZip2 [145], a multi-threaded scientific kernel MatMul, 13 programs in Splash2X [146] benchmark suit, and all 13 programs in Parsec [147] benchmark suit. The experiments show: 1) JUPITER can greatly improve QoS. On average, the QoS of JUPITER is 1.4x higher than three related systems. 2) JUPITER can improve system throughput. On average, JUPITER can improve system throughout by 50.8% compared with three related systems.



(a) Multi-threaded/Single-thread streamcluster exe. time (b) CPU utilization of different mixed multi-threaded workloads (c) Existing coscheduling approaches are not effective

Figure 4.1 Existing coscheduling approaches are not effective to mitigate the performance vulnerability problem for multi-threaded workloads. "p." and "s." mean programs from Parsec and Splash2X benchmark suits, respectively.

4.2 Background and Motivation

4.2.1 Coscheduling

Time-sharing CPU cores reduces the efficiency of multi-threaded applications due to the interplay of three factors: frequent synchronizations and communications between threads, work-conserving scheduling, and unfairness of conventional OS schedulers for multi-threaded applications. Specifically, in a multi-threaded application, threads need to frequently wait at synchronization/communication points. Under conventional work-conserving schedulers, a waiting thread will be descheduled to let the core be used for running other threads. However, when this waiting thread becomes ready to make progress, it may not be immediately scheduled. Such a delay not only postpones the progress of this particular thread, but also causes a “chain effect” through the synchronizations and communications between threads of the same application. For example, other collaborating threads will also be delayed, waiting for the data produced by this thread. Since conventional OS schedulers cannot effectively enforce fairness for multi-threaded applications, such waiting time can be significantly lengthened, exacerbating the unfair performance degradation of the whole application.

Coscheduling was proposed to deal with this problem for multi-threaded applications by co-running collaborating threads. By maximizing co-running, when a thread reaches a synchronization/communication point, the required data is either ready for use or will be produced by the collaborating threads very soon. In addition to reducing the waiting time for data, coscheduling can also reduce the waiting for cores. After a thread finished waiting for data at a synchronization/communication point, it can immediately be executed to make progress, since threads of this application are coscheduled.

Strictly co-running the collaborating threads can be implemented by using a non-work-conserving scheduler, but this can considerably reduce system throughput. Therefore, existing coscheduling approaches usually choose to boost the priority of the threads that synchronize/communicate frequently, so that they are less likely to be preempted by other threads during the co-running. However, such solutions are no longer effective in multi-tenant clouds where the multi-threaded applications are executed within the VMs.

4.2.2 Motivating Experiments and Goals

In this subsection, we conduct a few experiments with real-world applications on coscheduling to demonstrate that the performance issues persist on VMs, which promise isolated and dedicated execution environments.

Our experiments have two VMs consolidated on the same physical server. We run one multi-threaded benchmark in each VM, where we fix one VM to run `streamcluster` and the other to run different benchmarks in different experiments. All benchmarks are from `Parsec` [147] and Splash2X [148] benchmark suites. Section 4.5 gives the detailed configuration and settings.

We evaluate the coscheduling approach designed for VMs [149], namely balance scheduling, because it can avoid the performance degradation caused by the vCPU stacking problem [59]. We examine two settings: 1) *single-thread setting*, where each benchmark runs sequentially with one thread in a single-vCPU VM, and the two VMs share one core; and 2) *multi-threaded setting*, where each benchmark runs in parallel with 16 threads in a 16-vCPU VM, and the two VMs share the 16 cores.

Figure 4.1(a) shows that when `streamcluster` is collocated with different benchmarks, the multi-threaded executions suffer significant performance variation with the execution time varying widely between 171s and 493s. Some of the execution times are exceptionally long, indicating that the performance of `streamcluster` is very low. For instance, the execution time of `streamcluster` is 348s$\sim$493s when it is collocated with `fft`, `radiosity`, or `raytrace`. This is 5.5x$\sim$7.8x slow down, relative to the solo execution of `streamcluster` (i.e., no other VMs collocated with the VM running `streamcluster`). The slowdown is much higher than the reduction of the CPU time that `streamcluster` is entitled to use, 2x, i.e., from 100% of the system CPU time for its solo execution to 50% when collocated with another benchmark. This indicates that vulnerability of `streamcluster` to time sharing is high[1].

Figure 4.1(b) shows the CPU utilizations[2] of `streamcluster` and the benchmarks colocated with it. This can verify that the low performance is caused by `streamcluster` being unfairly penalized. When `streamcluster` is collocated with `fft`, `radiosity`, or `raytrace`, its CPU utilization is far below 50%, the fair share of CPU time that each benchmark is entitled to use, and `fft`, `radiosity`, and `raytrace` have CPU utilization much higher than 50%.

Figure 4.1(b) also indicates that the vulnerability of multi-threaded applications reduces the system throughput. For instance, for the mixed workloads of `streamcluster` and `raytrace`, the total CPU utilization is below 70%. However, when `streamcluster` or `raytrace` runs in solo, its CPU utilization is much higher than 50%. The system throughput may be reduced because, when a multi-threaded workload is unfairly penalized and cannot fully utilize its share of CPU time, the unused CPU time may be wasted if other workloads in the system cannot utilize it either. Our evaluation (§4.5) confirms that the system throughput of the mixed workloads can be improved through mitigating the performance vulnerability problem.

To show that the performance vulnerability is a particular problem of multi-threaded applications, Figure 4.1 (a) compares the performance degradation and performance variation of `streamcluster` in the multi-threaded setting against those in the single-thread setting. The execution times of `streamcluster` in the single-thread setting

[1]In our experiments, main memory is not over-committed; and the benchmarks have minimal I/O operations. Thus, the contention for other types of resource is not a major factor.

vary from 1018s to 1055s. Compared to the execution time of `streamcluster` when it runs alone, the execution times are increased by 2.3x$\sim$2.4x, which are roughly corresponding to the reduction of CPU time available to `streamcluster`. This indicates that `streamcluster` shows more robust performance when executed as a single thread than that with multi-threaded executions.

Coscheduling can reduce the performance vulnerability of multi-threaded applications through corunning collaborating threads. However, its effectiveness is seriously limited in existing co-scheduling schemes, and multi-threaded applications may still suffer serious QoS issues. To illustrate this, we repeat the experiments with balance scheduling [149] and two different coscheduling schemes in the VMM to manage vCPUs, including conventional coscheduling [11] and demand-based coscheduling [141; 150]. Since balance scheduling and conventional coscheduling are not open source, we implemented them in Linux/KVM based on their papers. Balance scheduling does not force the vCPUs in the same VM to be scheduled simultaneously; it only balances them onto different cores to avoid worst performance scenarios, in which multiple vCPUs from the same VM are stacked on the same core. With balance scheduling, we want to show the normal performance of multi-threaded applications without co-running. Conventional coscheduling and demand-based coscheduling show the performance of multi-threaded applications with strict co-running and less strict coscheduling, respectively.

Figure 4.1 (c) shows the execution times and CPU utilizations of `streamcluster` when it is collocated with different other applications. The experiments do confirm the effectiveness of corunning collaborating threads. For instance, when `streamcluster`

[2]CPU utilization of a benchmark here refers to the proportion of CPU time utilized by the VM running the benchmark.

is collocated with `fft`, `radiosity` or `raytrace`, demand-based coscheduling and conventional coscheduling can reduce the execution time by 17.5% and 22.8% on average, relative to that with balance scheduling. The average CPU utilization of `streamcluster` is increased from 11.3% with balance scheduling to 20.7% with demand-based coscheduling and 22.4% with conventional coscheduling, indicating that the unfair penalty on `streamcluster` is mitigated.

However, Figure 4.1 (c) also clearly shows that the effectiveness of corunning is seriously limited. When collocated with `fft`, `radiosity` or `raytrace`, the performance of `streamcluster` is still unfairly penalized, with CPU utilization still substantially lower than 50%; compared to the solo executions, the execution time of `streamcluster` is lengthened by 4.8x$\sim$6.6x for demand-based coscheduling and 4.5x$\sim$7.1x for conventional coscheduling.

The objective of the paper is to minimize the vulnerability of multi-threaded applications to the contention for CPU time when CPU cores are time-shared, so as to substantially improve the Quality of Service (QoS) for these applications, including preventing low performance, improving fairness, and increasing performance predictability. Another objective of the paper is to improve system throughput when systems are dominated with multi-threaded applications.

With a work-conserving scheduler, when an application is unfairly penalized and cannot fully utilize its CPU time share, other applications collocated with it may be unfairly rewarded, and obtain extra CPU time to achieve better performance. Improving the performance of the application may decrease the performance of other applications.

The paper considers such performance decreasing is favorable, since fairness and system throughput may be improved.

For instance, when `streamcluster` is collocated with `vips`, because `vips` cannot effectively utilize its CPU time share, `streamcluster` can utilize 75% of the CPU time in the system and reduce the execution time to 117s (shown in Figure 4.1 (a)). However, the better performance is achieved at the cost of fairness. In the example, `vips` is unfairly penalized, and its CPU utilization is only 11%. Thus, the performance of `vips` should be improved, though this may increase the execution time of `streamcluster`. For the collocation of `streamcluster` and `vips`, our solution aims to make `streamcluster` and `vips` fairly share CPU time and both achieve decent performance.

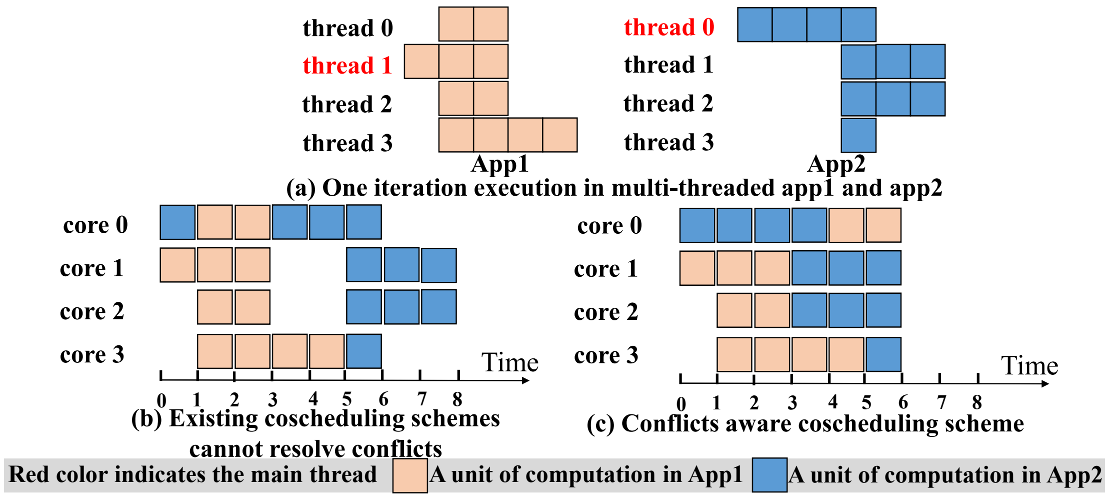
4.3 Main Ideas on Improving Coscheduling

This section identifies three key factors affecting the effectiveness of coscheduling. Then, it explains the directions and main ideas on substantially improving coscheduling, which are motivated by these factors.

• Factor 1: Conflicting Resource Demand for Boosting Multiple Applications

There lacks a mechanism to resolve conflicting resource demand for boosting multiple applications. When the threads of an application need to be coscheduled, coscheduling needs to boost the priority of these threads to facilitate them to corun. For brevity, the paper refers to these threads **prioritized threads**. When a system has two or more applications that rely on coscheduling to maintain high performance, it is possible that multiple prioritzed threads are scheduled together on the same core, and cause conflicts. The conflicting threads contend for the core, in order to maximize their corunning with the threads on other cores that are collaborating with them. Since not all the conflicting threads can be boosted, without a mechanism to resolve the conflicts, the effectiveness of coscheduling are counteracted.

Even worse, existing coscheduling approaches indiscriminately prioritize the threads to be coscheduled in each application. This unnecessarily increases the number of prioritized threads and the likelihood of conflicts. In some cases, this can even make conflicts unavoidable.



thread 0
thread 1
thread 2
thread 3
App1
thread 0
thread 1
thread 2
thread 3
App2
(a) One iteration execution in multi-threaded app1 and app2
core 0
core 1
core 2
core 3
Time
0 1 2 3 4 5 6 7 8
(b) Existing coscheduling schemes cannot resolve conflicts
core 0
core 1
core 2
core 3
Time
0 1 2 3 4 5 6 7 8
(c) Conflicts aware coscheduling scheme
Red color indicates the main thread
A unit of computation in App1
A unit of computation in App2

Figure 4.2 Existing coscheduling approaches are not effective for multiple applications that need coscheduling simultaneously.

Figure 4.2 explains this problem. Figure 4.2 (a) shows execution of a single iteration for a 4-thread application (denoted by *App1*), and another single iteration for another 4-thread application (denoted by *App2*). For each application, main thread (thread 1 in App1, and thread 0 in App2) generates and distributes tasks to other threads; when the tasks finish, the iteration ends, and main thread continues to generate and distribute tasks in a new iteration.

Figure 4.2 (b) illustrates the problems caused by the lack of a mechanism to resolve conflicts and indiscriminately prioritizing threads. The subfigure shows the execution of one iteration of these two applications when they collocated on a 4-core computer. Coscheduling tries to prioritize all the eight threads. A conflict happens on core 0 between thread 0 of App1 and thread 0 of App2 when time is 1. If the conflict is not resolved correctly, and the core is given to thread 0 of App1, thinking that all the other threads in App1 are running, the execution of App2 will be significantly delayed, and it takes eight units of time to finish these two iterations.

In Figure 4.2 (c), we assume that a resolving mechanism can correctly determine that thread 0 of App2 takes the core when the conflict happens. It takes only six units of time to finish the iteration. Compared to the schedule in Figure 4.2 (b), the schedule in Figure 4.2 (c) significantly improves the performance of App2 and system throughput; it might also improve the performance of App1: in Figure 4.2 (c), the second iteration of App1 can start from time is 6, but in Figure 4.2 (b) the second iteration can start from time is 6 only when thread 1 of App2 is migrated to core 0 or core 3.

• Factor 2: High Adverse Effects and High Cost to Reduce Adverse Effects

Existing coscheduling approaches enforce corunning aggressively by trying to schedule prioritized threads as early as possible and without interruption. This makes coscheduling notorious for its adverse effects and overhead. For instance, with existing coscheduling approaches, threads being coscheduled may aggressively preempt other running threads, and this increases context switches overhead. When threads being coscheduled run on the cores, they cannot be interrupted; thus these cores cannot handle the tasks with real high priority (e.g., latency-sensitive tasks), causing the priority inversion

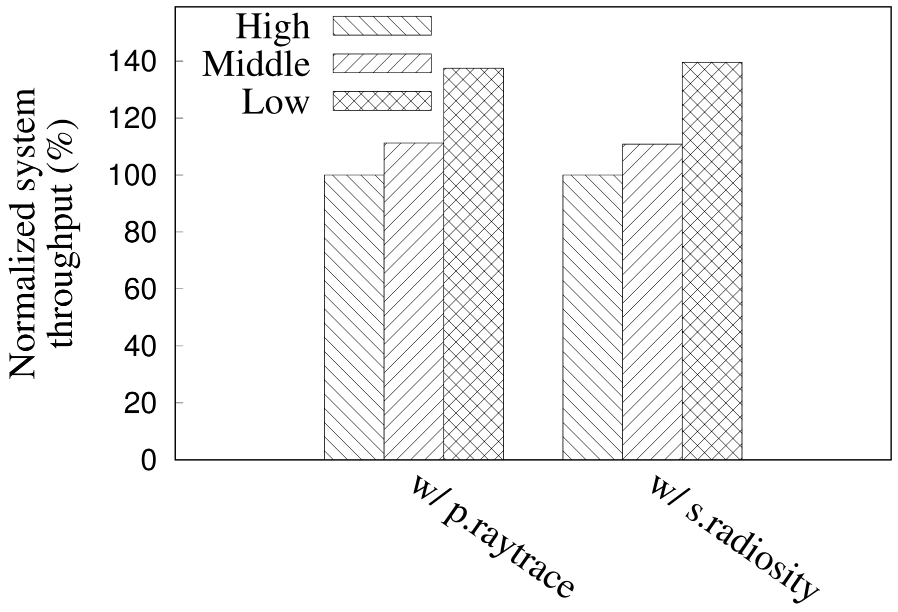


Figure 4.3 The lower system throughput can be caused by the adverse effects of coscheduling. The higher relax levelness of relaxed coscheduling, the lower system throughput. The system throughput is normalized to the system throughput when the relax level is high. `Streamcluster` is tested and collocated with `raytrace` or `radiosity`.

problem. When the collaborating threads of one application cannot use all the idle cores, and the remaining idle cores are enough to support other threads to corun, the CPU fragmentation problem is caused [149].

Existing coscheduling approaches may unnecessarily sacrifice the effectiveness when trying to reduce the adverse effects. For instance, to reduce CPU fragmentation, relaxed coscheduling [138] allows users to adjust the aggressiveness of coscheduling; a higher relax level makes coscheduling less aggressive by allowing a smaller portion of threads to start corunning. Though reducing CPU fragmentation helps improving system throughput, we show in Figure 4.3 that the system throughput is actually reduced (i.e., 38.5% on average) when making coscheduling less aggressive, because the effectiveness of coscheduling is reduced and the performance of multi-threaded applications is reduced as well. Relaxed coscheduling is evaluated with a commercial hypervisor through adjusting the relax level from high to low.

• Factor 3: Time Varying Computation Workload

Coscheduling boosts the priority of the threads within short time periods, which are determined by the available time slice of these threads. Thus, it cannot deal with the waiting periods longer than the time slices on the threads. Such long waiting periods reduce the computation workload in those periods and are reflected by the workload changing over time. Such workload changes are usually penalized, because scheduler usually provisions time slice periodically at a fixed rate, and unused time slice in the periods with light workload cannot be used later when workload is heavy. This “use it or lose it” policy is mainly to prevent an application from accumulating much time slice and later causing starvation to other applications when it intensively uses the time slice.

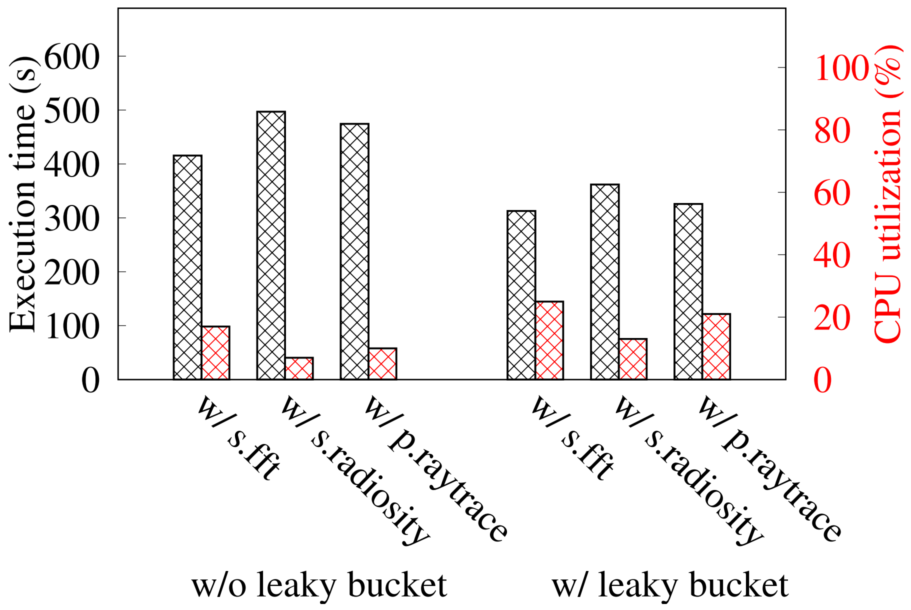


Figure 4.4 Leaky bucket is effective in mitigating performance vulnerability.

To justify the necessity to deal with this issue, we implemented a simple leaky bucket mechanism in Linux/KVM, which allows an application to accumulate a certain amount of unused time slice and use the accumulated time slice. We rerun the benchmark combinations in §4.2 using balance scheduling with this leaky bucket mechanism. Figure 4.4 compares the performance of `streamcluster` when it is collocated with `fft`, `radiosity`, or `raytrace`, and confirms that its workload changes do contribute to its performance vulnerability.

• **Main Ideas**

The above key factors motivate us to improve coscheduling from three directions. First, *prioritize threads in an asymmetric way*. This is to reduce/resolve conflicts, and also to reduce adverse effects. As the paper explains earlier, the indiscriminate prioritization increases both conflicts and adverse effects. However, there is no necessity to boost the priority of all the threads to the same high level. For instance, delaying the scheduling of some threads will not delay the execution of other threads or the waiting time of these threads at synchronization/communication points. Instead, the delay of scheduling such threads reduces or resolves conflicts by yielding resource to other threads that really need high priority. This can be illustrated using thread 0 in App1 in Figure 4.2 (b); as long as the thread can finish its task before time is 5, delaying its execution will not increase the waiting time of thread 1, since thread 3 cannot finish before time is 5. Delaying the execution of thread 0 in App1 resolves the conflict and allows thread 0 in App2 to utilize the resource to reduce the waiting time of other threads in App2. In addition, the comparison between Figure 4.2 (b) and (c) shows that further delaying the execution of the thread reduces the number of context switches (i.e., from two context switches on core 0 in Figure 4.2 (b) to one context switch in Figure 4.2 (c)).

Second, *prioritize threads less aggressively to reduce the adverse effects of coscheduling without reducing its effectiveness.* As we have shown above using thread 0 in App1 as an example, when reducing the aggressiveness of coscheduling (i.e., delaying the scheduling of a thread in the example), the adverse effects of coscheduling can be lowered. Moreover, if the reduction of aggressiveness is controlled within a certain range (i.e., the delay is not significant), the performance of the application is not degraded and the effectiveness of coscheduling is not affected. Thus, it is a natural idea to make coscheduling the least aggressive when keeping its effectiveness.

Third, *combine the leaky bucket technique and coscheduling to further improve the effectiveness of coscheduling.*

4.4 Design and Implementation

We implemented the main ideas described in §4.3 into JUPITER, which is a vCPU scheduler based on KVM. This section introduces the main challenges, overall design, and major components of JUPITER. Although the design is for vCPU scheduling, most parts of the design can be directly used in scheduling multi-threaded applications in multi-programming systems or containers. As explained in §3.1, we choose to implement our ideas in the VMM level for the following reasons. 1) VMs are prevalently used in clouds; 2) it is more challenging to implement the ideas at the VMM layer than other layers.

JUPITER consists of two parts, an Enhanced Leaky Bucket (ELB) mechanism and a JUPITER coscheduling mechanism. ELB is to periodically assign time slice to each VM in the system, and the JUPITER coscheduling mechanism is to schedule the vCPUs in each VM. We introduce the JUPITER coscheduling mechanism first, because the ELB mechanism needs the feedback from JUPITER coscheduling mechanism, and the JUPITER coscheduling mechanism is the major part.

4.4.1 JUPITER Coscheduling Mechanism

Overall Design and Challenges JUPITER implements the first two ideas introduced in §4.3. Thus, the problem it targets is essentially how to prioritize vCPUs in each VM in an asymmetric and unaggressive way, so as to 1) make the workload in the VM achieve the best performance with the CPU time assigned to the VM and 2) keep adverse effects low at the same time. JUPITER coscheduling mechanism addresses two challenges: how to effectively control the priorities of vCPUs, and how to achieve the above two goals?

To address the challenge with effective control of vCPU priorities, we identify/create a few system parameters that have the most influence on the relative progress of vCPUs, since the progress of vCPUs is the most important factor determining whether the vCPUs may spend much time on waiting for each other. Note that the priority used in JUPITER coscheduling mechanism is different from the system priority of the vCPUs (e.g., the "nice" values in Linux systems), and we choose to not use the system priorities in our coscheduling mechanism for two reasons: 1) they are used by the system for other purposes, which we do not want to mess up; and 2) they cannot provide the fine-grained control over the relative progress of vCPUs. In our design, we choose the following parameters: 1) *rescheduling latency* to control the time that the computation starts. In system designs, rescheduling latency is the parameter determining when a vCPU can be scheduled after it becomes "ready". In some systems (e.g., Linux), there is a system-wide rescheduling latency for all the vCPUs; we need to modify the system to create a private rescheduling latency for each vCPU. 2) *time slice* of a vCPU to control how much progress a vCPU can make after it is scheduled to run. In some rare cases, when the scheduler finds that these two parameters cannot effectively control the priority of a vCPU, it also checks whether the execution of the vCPU may be interrupted by other vCPUs and takes this factor into account.

To address the second challenge, we use a step-by-step iterative method to adjust the parameters above to approach to the two goals, i.e., high performance and low adverse effects. Specifically, for the first goal, it is not possible to directly measure the end-to-end performance of the workload in a VM. Instead, we use the CPU time consumed by the VM as an indicator of the amount of progress made by the workload. This indicator is reliable because the CPU time consumed by effective computation is roughly proportional to the amount of finished computation, idling does not consume CPU time, and vCPUs that perform excessive busy waiting are preempted promptly and consume little CPU time. For the adverse effects, it is not realistic either to measure it in practice. Thus, instead of measuring it, we lower the priorities of the vCPUs under the condition that lowering the priorities will not reduce workload performance.

There are two challenging issues with adjusting the parameters to achieve the goals. One is whether the priority of the vCPU should be higher or lower, and the other one is which parameter should be adjusted. There are a few design choices on adjusting the parameters. For instance, the parameters can be adjusted based on the CPU utilization of the workload. If CPU utilization has been maximized within the CPU share of the VM, we adjust the parameters to lower the priorities; otherwise, we adjust the parameters to improve CPU utilization. However, this design has a scalable issue to maintain a global CPU utilization for each VM, which might be high if each VM has a large number of vCPUs.

We choose to use a more scalable way, which adjusts the parameters of each vCPU based on the status of the vCPU: if increasing the priority of the vCPU helps improving performance, we increase the priority; otherwise, we reduce the priority to reduce adverse effects. We check the status of the vCPUs at the beginning of each period when the VMM is about to allocate new time-slice to the vCPUs of a VM, and based on the status adjust its parameters gradually. For example, if a vCPU has fully utilized its time-slice in the previous period and the vCPU is in a "ready" state, meaning that it could have made more progress if there were more CPU time, we will allocate more CPU time in the coming period to improve performance. We introduce the detailed design and the components for adjusting the parameters below.

Design and Implementation Details The implementation of JUPITER coscheduling mechanism includes three key components: (1) a time-slice adjustment component for Dynamically Adjusting the Time-Slice (DATS) distribution between the vCPUs of each VM, (2) a Dynamically Adjusting reScheduling Latency (DASL) component, and (3) a Resource Conflict Resolver (RCR). The first two components reduce conflicts and the last component detects and resolves the conflicts that cannot be reduced by the first two components. For the first two components, we focus on introducing how the adjustment decisions are made, since enforcing the decisions is straight-forward and system-dependent.

The time-slice allocation component collects the amount of time-slice consumed by each vCPU periodically and uses the amount of time-slice consumed by the vCPUs in the previous period to adjust the amount of time-slice to be allocated to the vCPUs in the upcoming period. Specifically, for a vCPU that has been preempted earlier due to the depletion of time slice, the component increases its time-slice. For other vCPUs, since they still have unused time-slice at the end of the period, there is no need to further increase their time-slice. The component assigns a weight to each vCPU. To increase the time-slice of a vCPU, the component increases the weight by 10%. The component keeps the total weight of the vCPUs in a VM fixed. Thus, it reduces the weight of other vCPUs accordingly based on their original weights.

The rescheduling latency adjustment component looks at whether the vCPU has consumed its time-slice and whether the vCPU can still make progress at the end of each period. For a vCPU that is in a “ready” or “running” state at the end of previous period, the vCPU cannot consume its time-slice quickly. This may be caused by rescheduling delay. Thus, the component decreases the rescheduling latency of the vCPU by 10%. For the vCPUs that have consumed their time-slice and become idle at the end of previous period, the component increases their rescheduling latencies.

There are scenarios, in which a vCPU with a low rescheduling latency has tasks depending on the completion of the tasks on other vCPUs with high rescheduling latencies. Since the tasks on the vCPUs with high rescheduling latencies complete late, the task on the vCPU with a low rescheduling latency cannot start early. Thus, it is possible that the vCPU with a low rescheduling latency still cannot consume its time-slice, no matter how its rescheduling latency is reduced. To detect such scenarios, when the rescheduling latency of a vCPU has been reduced to a minimal value allowed by the system, if a vCPU still cannot consume its time slice, the component assumes that the vCPU may be delayed by other vCPUs with high rescheduling latencies. To pin-point these vCPUs, the component uses wake-up inter-processor interrupts (IPIs) sent to the vCPU as indicators to find out the source vCPUs sending out the IPIs. Then it reduces the rescheduling latencies of these source vCPUs.

Resource Conflict Resolver. The adjustment of time-slice distribution and rescheduling latencies of vCPUs effectively make the vCPUs having asymmetric and low priorities. They significantly reduce conflicts. However, conflicts cannot be completely avoided by these two components. The last component detects and tries to resolve such conflicts by migrating vCPUs between cores. For JUPITER coscheduling mechanism, conflicts arise when the total amount of time-slice allocated to the vCPUs scheduled on the same core exceeds the core's capacity. For example, a conflict arises when, in a time period of 80ms, each of two vCPUs scheduled on the same core is allocated with 50ms time-slice. The vCPUs with low rescheduling latencies may also have conflicts. A conflict arises when a core is running a vCPU with low rescheduling latency and another vCPU with low rescheduling latency becomes ready to run. If the former vCPU is preempted promptly, its task is essentially delayed since the task cannot be finished quickly. If the former vCPU is not preempted promptly, the latter vCPU cannot be rescheduled quickly.

RCR tries to resolve conflicts by adjusting the layout of vCPUs on physical cores. Since adjusting vCPU layout is costly, RCR does the adjustment in a conservative way. Specifically, to detect and resolve conflicts caused by high demands for CPU time, after the time-slice allocation component has adjusted the amounts of time-slice to be allocated to each vCPU, for each core, RCR calculates an aggregated amount of time-slice for the vCPUs scheduled on the core. Then, RCR finds out the core with the largest aggregated amount and the core with the smallest aggregated amount. If the largest aggregated amount is greater than the smallest aggregated amount by 10%, RCR tries to balance the aggregated amounts by swapping some of the vCPUs on the two cores.

To detect and resolve conflicts caused by the vCPUs with low rescheduling latencies, after the rescheduling latency adjustment component has adjusted the rescheduling latency of each vCPU, RCR categorizes the vCPUs into two groups based on their rescheduling latencies — vCPUs with low rescheduling latencies and vCPUs with high rescheduling latencies. In each period, RCR monitors the execution of the vCPUs with low rescheduling latencies. It counts the number of times that these vCPUs are preempted and the number of times that these vCPUs are not scheduled after they become ready and have waited a long time exceeding their rescheduling latencies. After the period, it uses the total number as the number of conflicts on the core caused by the vCPUs with low rescheduling latencies. Then, RCR finds out the core with the most conflicts and the core with the fewest conflicts. If the difference between the numbers of conflicts exceeds a threshold (2x in implementation), RCR selects half of the vCPUs with low rescheduling latencies on the core with the most conflicts and half of the vCPUs with high rescheduling latencies on the cores with the fewest conflicts, and then swaps the vCPUs. Low thresholds increase vCPU migration overhead. High thresholds "cripple" the conflict resolver. Thus, we measured how vCPU migrations reduce with increased thresholds, and selected the threshold values at knee points to make trade-off.

4.4.2 Enhanced Leaky Bucket

In time-sharing environments, "loan and borrow" is a commonly used and effective variant of the leaky bucket mechanism. We also choose to integrate the "loan and borrow" method with the JUPITER coscheduling mechanism. Here, we explain the special designs in the integration, and the details of the original method can be found in [151]. Specifically, the "loan" and "borrow" operations essentially change the time slice available to each application. Due to the performance vulnerability of multi-threaded applications, these operations may have a significant impact on performance and must be carried out carefully. In JUPITER, when a VM "loans" another VM CPU time, the amount is increased gradually, and the performance degradation (i.e., CPU utilization in our implementation) of the lending VM is closely monitored. If the degradation exceeds a threshold (5% in implementation), we stop increasing the amount of the "loan" to prevent significant performance degradation of the lending VM. At the same time, if there are multiple borrowing VMs, the lending VM chooses to "loan" the VMs which can generate the most significant performance improvement; thus, it refers to the RCR component to identify the VMs suffering the fewest conflicts.

4.5 Evaluation

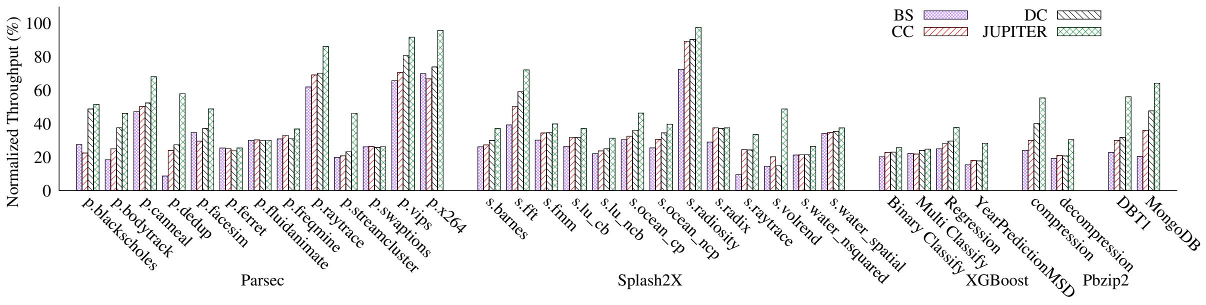


Figure 4.5 QoS under BS, CC, DC, and JUPITER systems in the homogeneous setting: higher normalized throughput means better QoS.

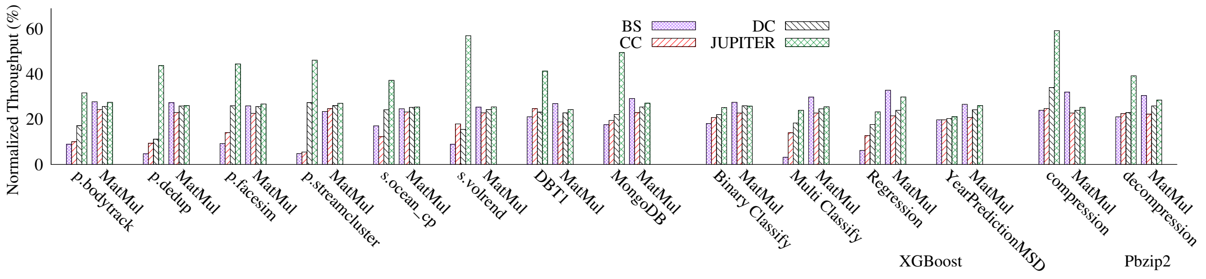


Figure 4.6 QoS under BS, CC, DC, and JUPITER systems in the heterogeneous setting: higher normalized throughput means better QoS.

We evaluated JUPITER by answering the following questions. 1) How much can JUPITER improve QoS compared to prior systems (§4.5.1)? 2) How much can JUPITER improve system performance (§4.5.2)? 3) Can JUPITER consistently improve QoS across different VM sizes (§4.5.3)? 4) How effective is each technique in JUPITER (§4.5.4)? 5) Can JUPITER efficiently handle the adversarial workload? What is JUPITER's overhead (§4.5.5)?

Experimental setup. We conducted experiments on a DELL^{TM} PowerEdge^{TM} R720 server with 64GB of DRAM and two 2.40GHz Intel® Xeon® E5-2665 processors. Each processor has 8 cores. We created two or four VMs using Linux KVM [143], where each VM has 16 vCPUs and 16GB memory. Both the host OS and guest OS were Ubuntu 16.04 with the Linux kernel version 4.19.1. The vCPUs in each VM were laid out on the cores in a way to prevent the vCPU stacking problem [149; 152].

We implemented JUPITER in Linux KVM and evaluated it on 7 real-world applications, including 2 database server programs `PgSql` [114] and `MongoDB` [24], classical programs in `Parsec` [147] and Splash2X [146] benchmark suites, 4 widely used AI programs in `XGBoost` [153], a TPC-W like benchmark from the OSDL database test suite `DBT1` [116], a parallel compression utility PBZip2 [145], and a parallel CPU-bound program `MatMul`.

Our experiments were conducted under two main settings: 1) homogeneous setting where VMs ran the same workload; and 2) heterogeneous setting where VMs ran different workloads. We show the average results of five runs in all experiments.

We compared JUPITER with BALANCE SCHEDULING (BS) [149] and two representative coscheduling systems: CONVENTIONAL COSCHEDULING (CC) [11; 149] and DEMAND-BASED COSCHEDULING (DC) [141; 150]. We chose BS as the baseline because BS performs better than Vanilla KVM in our experiments, which is also confirmed in [149]. Because BS and CC are not open source, we implemented them according to their papers.

4.5.1 QoS Improvement

We first compared the QoS provided by BS, CC, DC, and JUPITER on various programs running in four 16-vCPU VMs consolidated on the server. Under the homogeneous setting, all VMs ran the same workload. Under the heterogeneous setting, one VM ran a synchronization-intensive workload, while the other three ran the same computation-intensive workload (e.g., `MatMul` or `PgSql`). Since the throughputs of different applications vary largely, we normalized the throughput of one application by dividing it with the throughput of the same application running in a VM on a dedicated server (i.e., the solo run). Hence, the larger the normalized throughput, the better the QoS.

Figure 4.5 shows the normalized throughputs of different applications under the four systems in the homogeneous setting. On average, the performance of JUPITER is 1.6x, 1.4x, and 1.3x higher than BS, CC, and DC, respectively.

In particular, JUPITER outperforms CC, because CC coschedules the coordinated threads only when there are enough available pCPUs, which incurs CPU fragmentation and extra context switches. Although DC can more quickly coschedule urgent vCPUs by monitoring the IPI information and increase the priority of the urgent vCPUs, these vCPUs may not use up their fair share of CPU time. This is because with multiple multi-threaded workloads there can be multiple prioritized vCPUs on the same core that contend on the CPU resource. Similarly, although BS is designed to avoid CPU fragmentation and increase the likelihood of coscheduling coordinated vCPUs by scheduling them on different physical CPUs, it is not able to properly handle the contention from multiple multi-threaded workloads.

The heterogeneous setting, shown in Figure 4.6, evaluates the amount of performance degradation incurred on the synchronization-intensive workloads, when collocated with computation-intensive workloads (i.e., `MatMul`). For synchronization-intensive workloads, we choose the six applications (i.e., `bodytrack`, `facesim`, dedup, `streamcluster`, `ocean_cp`, volrend) from `Parsec` and Splash2X benchmark suites, together with `XGBoost` and PBZip2 applications. The average normalized throughput of the synchronization-intensive workloads is 20.5%, 19.4%, 23.2% and 32.6% for BS, CC, DC and JUPITER, respectively. Meanwhile, the normalized throughputs of the collocated computation-intensive workload (i.e., `MatMul`) are comparable among different systems.

JUPITER achieves significantly better QoS than the other systems for synchronization-intensive applications, when collocated with computation-intensive workloads. This is because the threads waiting for the remaining threads to reach the synchronization cannot make any progress, even if their vCPUs have abundant remaining time slice. However, when co-run with computation-intensive `MatMul`, the unused CPU fair share of the synchronization-intensive application is used by `MatMul` without any compensation under BS, CC, and DC. In contrast, JUPITER solves this problem with `DATS` and `ELB`.

Next, we conduct similar homogeneous and heterogeneous experiments for latency-sensitive applications: TPC-W and `MongoDB`. For these applications, we normalize the measured response time by dividing it with the response time under the solo run. Hence,

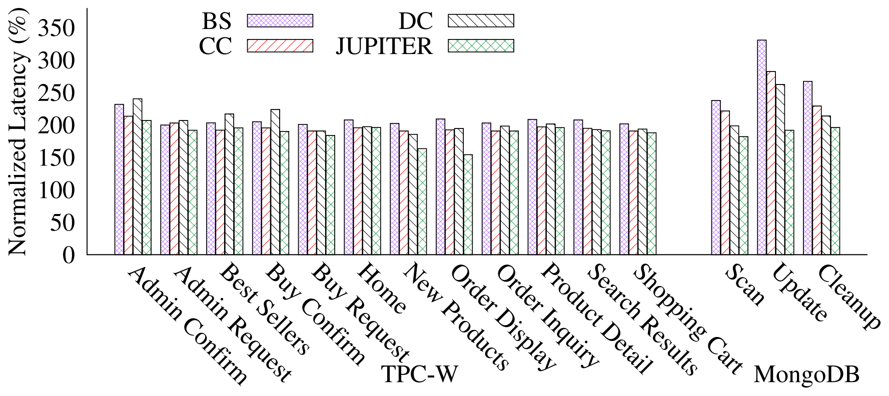


Figure 4.7 QoS for latency-sensitive applications in the homogeneous setting: lower normalized latency means better QoS.

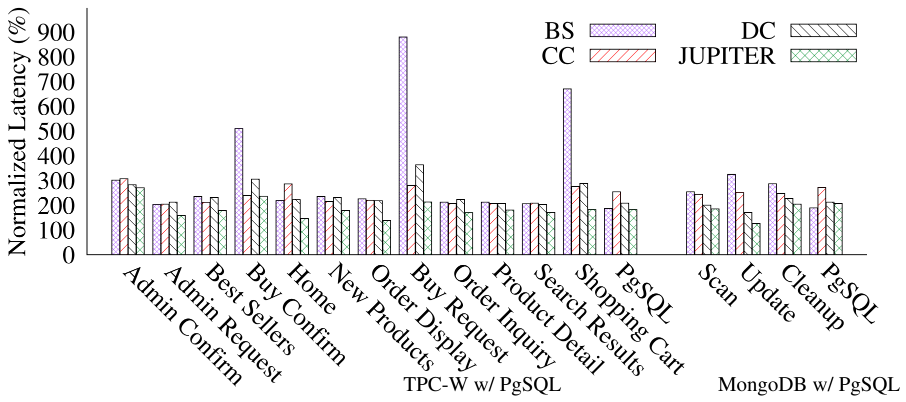


Figure 4.8 QoS for latency-sensitive applications in the heterogeneous setting: lower normalized latency means better QoS.

the lower the normalized latency, the better the QoS of the application. As shown in Figure 4.7, the average latencies under the heterogeneous setting are 207.2%, 192.4%, 194.7% and 176% for BS, CC, DC and JUPITER, respectively. The improvement of JUPITER again comes from the fact that it intelligently resolves the resource contention for coordinated threads of multiple multi-threaded workloads.

For the heterogeneous setting with the co-running computation-intensive `PgSql`, Figure 4.8 shows that the average latencies become 297.8%, 230.1%, 223.2% and 174.5% for BS, CC, DC and JUPITER, respectively. The latencies under BS drastically increase in the heterogeneous setting, because it does not ensure the co-running of synchronizing threads, which results in high synchronization overhead. This problem deteriorates when the collocated computation-intensive workloads can unfairly take advantage of that. The unfairness issue lessens under CC and DC. As JUPITER solves this problem with `DATS` and `ELB`, it effectively reduces the average latencies under BS, CC, and DC by 77.3%, 35.9%, and 33.7%, respectively.

4.5.2 System-Wide Improvement

The next experiments examine the overall system performance in the heterogeneous setting, with two 16-vCPU VMs consolidated on the server. We vary the application in one VM and co-run it with `freqmine` or dedup in the other VM, as they have different features. `Freqmine` is computation-intensive with less synchronization among threads, while `Dedup` is synchronization-intensive. We calculate the geometric mean of applications' speedups (i.e., weighted speedup) as a measure of the overall system performance. We normalized the weighted speedup to that of BS for different applications.

Figure 4.9 presents the results when the co-runner is `freqmine`. On average, the weighted speedup of JUPITER is 1.3x, 1.4x, and 1.2x higher than BS, CC, and DC, respectively. For most applications, CC has the worst performance, since it causes adverse effects (e.g., CPU fragmentation, execution delay, extra context switches) to the whole system. These adverse effects are much more severe when the co-runner is dedup with frequent synchronizations. As shown in Figure 4.10, the weighted speedup of CC is 49.2%, 46.1%, and 81.2% worse compared to BS, DC, and JUPITER, respectively.

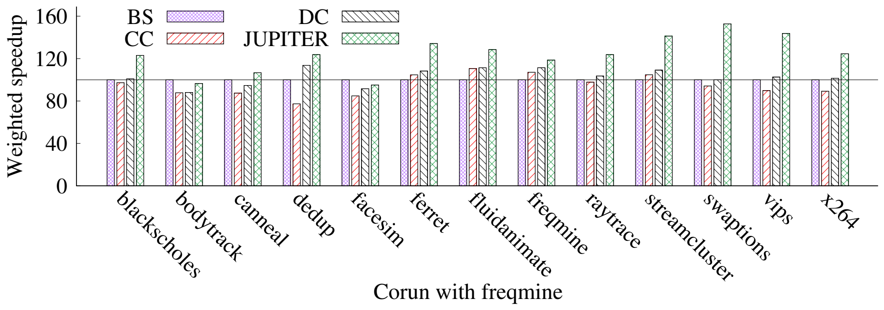


Figure 4.9 Weighted speedup when co-running with freqmine: higher weighted speedup means better system-wide performance.

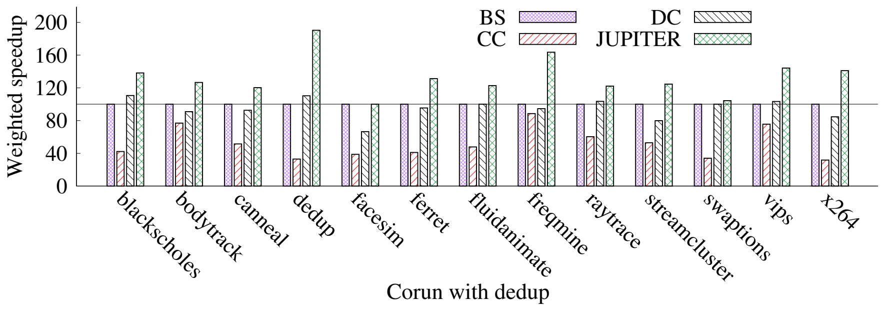


Figure 4.10 Weighted speedup when co-running with dedup: higher weighted speedup means better system-wide performance.

JUPITER has notably better system-wide performance than the other systems because it allows VMs containing multi-threaded applications to better utilize physical CPUs. For instance, when co-running with `freqmine`, the average CPU utilization under BS, CC, DC, and JUPITER is 74.3%, 61.7%, 81.2%, and 90.8%, respectively. We notice that, even under JUPITER, the CPU utilization is not close to 100% for some benchmarks. This is because some benchmarks have certain execution phases where the computation is sequential and cannot utilize all the CPUs.

Furthermore, JUPITER reduces performance variation when the co-running workload incurs different levels of interferences. On average, the standard deviation of weighted speedups is 11.3, 20.8, 12.6, and 6.2 under BS, CC, DC, and JUPITER, respectively. This is because JUPITER not only accumulates and compensates unused CPU fair share, but also allocates CPU time to vCPUs considering both their demands and resource conflicts.

Together with the results in Section 4.5.1, we can clearly see that JUPITER can improve system performance and QoS consistently, regardless of the collocated workloads.

4.5.3 Consistent Advantage across Different VM Sizes

Next, we set up 4 VMs and vary the number of vCPUs of each VM from two to 16. Figure 4.11 and Figure 4.12 show the performance of JUPITER and BS under the homogeneous and heterogeneous settings, respectively. Again, JUPITER achieves significantly higher normalized throughput than BS. In addition, the varying number of vCPUs does not affect JUPITER much, while BS has higher performance degradation given more vCPUs. This is because JUPITER not only accumulates and compensates the unused fair share of VMs, but also improves the ability of the synchronization-intensive workload to use up its fair share of CPU time. Specifically, it distributes the fair share of a VM to its vCPUs based on their demands and urgency, which allows faster execution of the critical thread and hence faster progress of the dependent vCPUs.

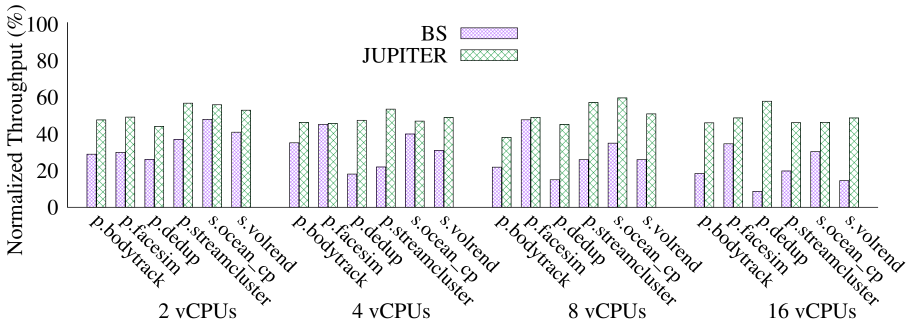


Figure 4.11 QoS with varying number of vCPUs under homogeneous setting.

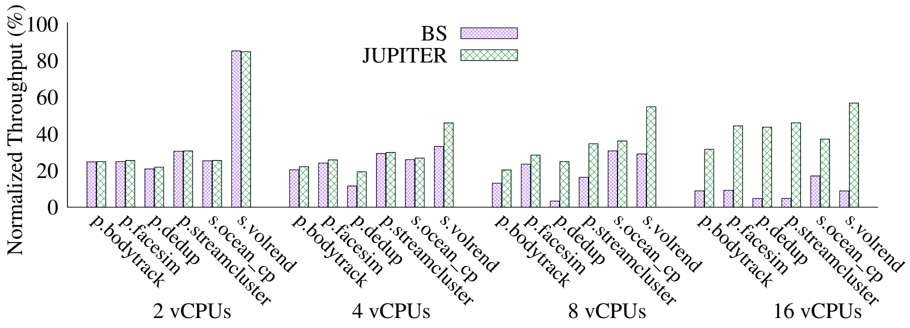


Figure 4.12 QoS with varying number of vCPUs under heterogeneous setting.

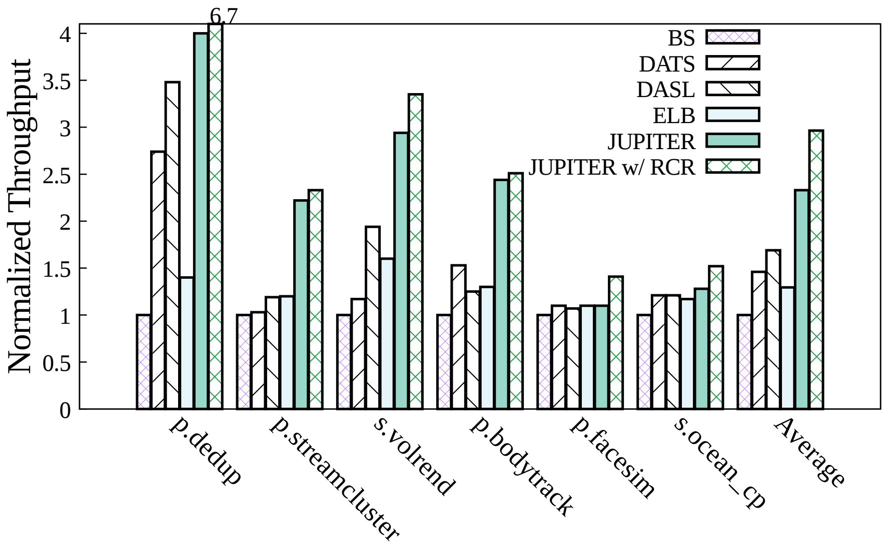


Figure 4.13 Performance analysis of JUPITER's different techniques.

4.5.4 Effectiveness of JUPITER's Each Technique

Figure 4.13 analyzes the improvement of JUPITER's different techniques. This experiment has four 16-vCPUs VMs running the same workload. The throughput is normalized to BS. On average, performance improvement is 46.3%, 69.1% and 29.5% for individual DATS, DASL and ELB, respectively. DATS and DASL have more improvement than ELB, indicating that only compensating the unused CPU fair share may be sufficient for achieving better QoS. This is because within one VM the critical vCPU may not be allocated with enough CPU time or be fast scheduled, so the dependent vCPUs cannot make progress even with an abundant CPU time. These three techniques work synergistically and improve performance by 2.33x compared with BS. RCR can further improve the performance of JUPITER by 33.5%, as the coordinated threads can co-run for a long time without being interrupted.

4.5.5 Robustness to Adversarial Workload

Finally, we evaluate the robustness of JUPITER on handling the adversarial workload. In Figure 4.14, there are two MongoDB clients at 0s. From 6s to 7s, three new MongoDB clients are added at the beginning of each time period (i.e., 100ms) and then killed at the beginning of the following time period. Initially, JUPITER cannot correctly predict the time amount for vCPUs due to the severe change in workloads, its throughput is slightly worse than that of BS. However, this lasts shortly. At 6.5s, when JUPITER finds that it cannot consecutively predict the time amount for vCPUs correctly, it is disabled automatically. In the disabled time periods, it still makes the prediction for the coming time periods, but all the decisions are not enforced to the system. From 6.6s to 7.0s, JUPITER's throughput

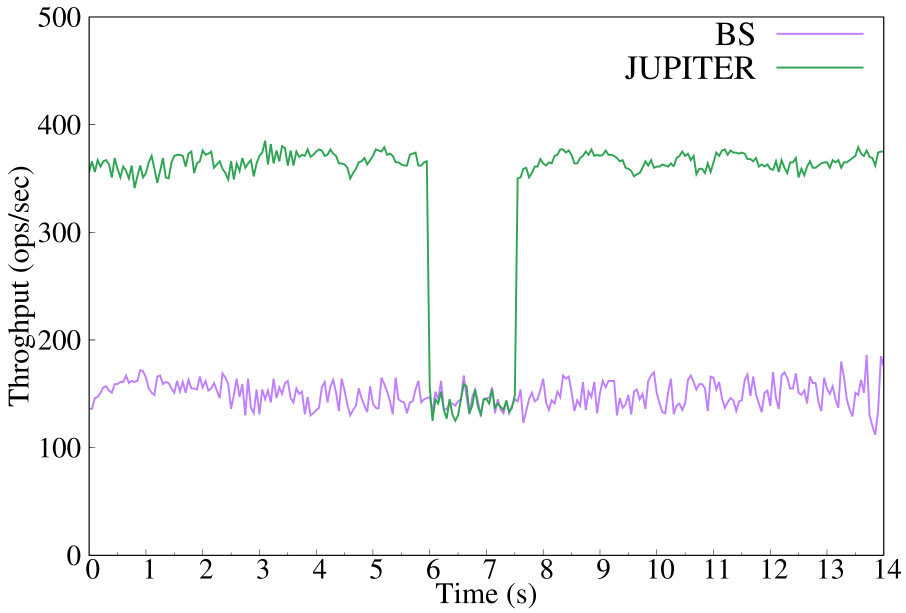


Figure 4.14 JUPITER's performance on handling adversarial workloads. Four 16-vCPU VMs are consolidated on the server. Each VM is running `MongoDB`.

is almost the same as BS's throughput. This shows JUPITER incurs low overhead (less than 3%). From 7.1s to 7.5s, JUPITER can again correctly predict the CPU time used by vCPUs. Thus, it automatically starts from 7.6s. JUPITER's throughput improvement over BS resumes.

4.6 Related Work

Co-scheduling. Co-scheduling [11; 154; 155] is a representative scheme for multi-threaded applications to mitigate synchronization delay by executing cooperative threads simultaneously. However, its original form that always co-runs the cooperative threads incurs high overhead (e.g., CPU fragmentation, execution delay, etc.) because threads cannot be scheduled until all their required CPUs become available. Hence, many coscheduling variants have been designed to mitigate such overhead.

Balance scheduling [149] increases the likelihood of coscheduling coordinated threads by scheduling them on to different CPUs. Demand-based coscheduling [136; 135; 136; 141; 140] dynamically applies coscheduling to coordinated threads, whereas non-communicating threads are managed in an uncoordinated fashion. Once it detects the coordination demand, it prioritizes coordinated threads via preempting the other threads running on CPUs. Relaxed coscheduling [138] mitigates CPU fragmentation through only coscheduling a subset of collaborating threads to prevent their runtime from being largely skewed. Similarly, flexible coscheduling [139; 134; 156; 157] only applies coscheduling to synchronization-intensive workloads to improve their performance. Meanwhile, uncoordinated workloads can fill CPU fragmentation caused by coscheduling to improve system throughput.

Existing coscheduling approaches are not effective to multiple collocated workloads that all need coscheduling simultaneously. Moreover, they do not consider QoS in multi-tenant clouds. In contrast, JUPITER is a general, efficient, and QoS-aware approach that can greatly improve QoS for collocated multi-threaded workloads in clouds.

CPU fair scheduling. Many existing systems focus on proportional-share scheduling [97; 158; 159; 160; 161; 158; 162; 163; 164; 165; 166; 151]. The basic idea is to assign CPU resources to applications based on their CPU share or weight. However, these works do not consider synchronization among threads, nor do they consider QoS in multi-tenant environments. Moreover, they allow threads' CPU allocations to be forfeited without compensation. For instance, completely fair scheduler [97] and credit scheduler [163; 164] improve the priority of awoken threads by withdrawing part of their credits, which can cause CPU fairness problems.

For multi-tenant clouds, FLEX [167] focuses on the CPU fairness problem among VMs when they have different sizes (i.e., number of vCPUs). It improves fairness among VMs by dynamically tuning the weights of VMs. Preemptive multi-queue fair queuing (P-MQFQ) [168] improves fairness among programs through preempting the programs that use more CPU fair share. However, FLEX and P-MQFQ do not consider the fairness problems among multiple collocated workloads that need to be prioritized simultaneously. In particular, they cannot handle resource conflicts when these workloads need to be prioritized to get their CPU fair share at the same time.

Other works on improving QoS in clouds includes resource provisioning [3; 4; 5; 169; 170; 12; 171; 172; 89; 173; 174] and VM placement (i.e., admission control) [6; 7; 175; 176; 88; 13; 177]. These works mainly consider QoS in coarse-grain, which is orthogonal to JUPITER.

XEN and Co. [133] presents workloads perfromance degradation caused by rescheduling delay in XEN hyppervisor (e.g., Domain0 is not scheduled on time to process send-out or received packets), and it develops a communication-aware CPU scheduler, expecting to schedule delayed domains timely and fairly to mitigate such delay and improve performance.

4.7 Conclusion

The paper addresses the critical issue of poor QoS, unfairness, and low performance for running multi-threaded applications in clouds. Existing coscheduling-based solutions for multi-threaded applications are no longer effective for multi-tenant cloud environments. The work identifies three reasons that make coscheduling ineffective for VMs, proposes a new QoS-aware approach, and tests with extensive experiments. Our solution can significantly improve the QoS of multi-threaded applications running in standalone servers. In the future, we plan to design the corresponding admission control strategies that can distribute workloads smartly to a cluster of servers to provide better QoS guarantees.

CHAPTER 5

CONCLUSION

Multi-threaded applications of different tenants remain notoriously difficult to achieve QoS in clouds. These applications often suffer severe performance degradation and interference, scalability problems, fairness issues, and so on.

To improve the multi-threaded QoS in clouds, we have identified three major reasons and invented three systems, VMIGRATER, VSMT-IO, and JUPITER, with each addressing a distinct research problem. We have shown that our systems can greatly improve the QoS of multi-threaded applications.

As future work, we plan to extend our systems to wider scenarios. For instance, we want to extend the design of VSMT-IO to processors with more hardware threads, test if more thoroughly, and seek the adoption in real systems utilized in industry.

REFERENCES

[1] "AWS T2 Instances," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html, 2018.

[2] "Amazon EC2 Instance Types," https://aws.amazon.com/ec2/instance-types/, 2018.

[3] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 289–302.

[4] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 13–26.

[5] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 237–250.

[6] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of the 2013 USENIX Annual Technical Conference*, ser. USENIX ATC'13, no. EPFL-CONF-185984, 2013.

[7] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48. ACM, 2013, pp. 77–88.

[8] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 4.

[9] E. Amazon, "Amazon elastic compute cloud: User guide," 2013.

[10] VMWare, "Vmware resource pools," https://docs.vmware.com/en/VMware-vSphere/5.5/vsphere-esxi-vcenter-server-552-resource-management-guide.pdf.

[11] J. K. Ousterhout *et al.*, "Scheduling techniques for concurrebt systems." in *ICDCS*, vol. 82, 1982, pp. 22–30.

[12] C. Delimitrou and C. Kozyrakis, "HCloud: Resource-Efficient Provisioning in Shared Cloud Systems," in *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[13] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167.

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

[15] RedHat, "What is the suggested I/O scheduler to improve disk performance (2017)." https://access.redhat.com/solutions/5427.

[16] 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*. ACM, 2012, pp. 3–14.

[17] 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*. ACM, 2017, pp. 269–281.

[18] "KVM Virtual Machine Monitor," http://www.linux-kvm.org/.

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

[20] "The Hadoop Distributed File System," http://hadoop.apache.org/hdfs/.

[21] "HBase," https://hbase.apache.org/.

[22] "The PostMark Benchmark," http://www.filesystems.org/docs/auto-pilot/Postmark.html.

[23] "LevelDB," https://github.com/google/leveldb.

[24] "MongoDB KV Storage Benchmarks," http://www.mongodb.org, 2012.

[25] https://en.wikipedia.org/wiki/Completely_Fair_Scheduler.

[26] https://en.wikipedia.org/wiki/Deadline_scheduler.

[27] https://en.wikipedia.org/wiki/Noop_scheduler.

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

[29] 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*, vol. 35. ACM, 2001, pp. 117–130.

[30] https://en.wikipedia.org/wiki/CFQ.

[31] 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*, 2013, pp. 243–254.

[32] 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*. IEEE Computer Society, 2014, pp. 394–405.

[33] 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*. ACM, 2009, pp. 101–110.

[34] A. Kivity, D. L. G. Costa, and P. Enberg, "Os v—optimizing the operating system for virtual machines," in *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*, 2014, p. 61.

[35] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ACM SIGPLAN Notices*, vol. 48. ACM, 2013, pp. 461–472.

[36] 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*. ACM, 2008, pp. 1–10.

[37] B. Teabe, A. Tchana, and D. Hagimont, "Application-specific quantum for multi-core platform scheduler," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 3.

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

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

[40] https://lkml.org/lkml/2017/2/14/733.

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

[42] https://iovisor.github.io/bcc/.

[43] B. Gregg, "Performance superpowers with enhanced BPF," in *USENIX ATC 2017*. Santa Clara, CA: USENIX Association, 2017.

[44] "Redhat Technical Notes," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/6.0_technical_notes/deployment.

[45] RedHat, "Redhat performance tuning guide," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/index.

[46] "MediaTomb - Free UPnP MediaServer," http://mediatomb.cc/, 2014.

[47] "Nginx web server," https://nginx.org/, 2012.

[48] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004, pp. 10–10.

[49] "Clam AntiVirus Benchmarks," http://www.clamav.net/.

[50] 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*. USENIX Association, 2012, pp. 2–2.

[51] "Apache Hadoop Systems," http://hadoop.apache.org/core/.

[52] "LevelDB Benchmarks," http://www.lmdb.tech/bench/microbench/benchmark.html.

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

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

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

[56] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, "Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications." in *USENIX Annual Technical Conference*, 2014, pp. 73–84.

[57] 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*. ACM, 2007, p. 4.

[58] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007, p. 2.

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

[60] 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*. ACM, 2016, p. 2.

[61] 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*. ACM, 2012, p. 2.

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

[63] 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*. IEEE, 2010, pp. 1–11.

[64] 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*. ACM, 2015, pp. 125–136.

[65] 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*. ACM, 2014, pp. 1–13.

[66] 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*. IEEE, 2015, pp. 453–460.

[67] 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*. ACM, 2011, p. 24.

[68] 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*. ACM, 2015, pp. 125–138.

[69] 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*. ACM, 2011, p. 47.

[70] 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.

[71] "Amazon EC2 Instance Types," https://aws.amazon.com/ec2/instance-types/#instance-details, 2018.

[72] "SMT Configurations in VMWARE," https://bit.ly/1LxQTiW.

[73] "Introducing hyperthreading into azure vms," https://azure.microsoft.com/en-us/blog/introducing-the-new-dv3-and-ev3-vm-sizes/.

[74] "Google Cloud Virtual Machine Types," https://cloud.google.com/compute/docs/machine-types.

[75] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 234–244, 2000.

[76] J. Nakajima and V. Pallipadi, "Enhancements for hyper-threading technology in the operating system: Seeking the optimal scheduling." in *WIESS*, 2002, pp. 25–38.

[77] K. Deng, K. Ren, and J. Song, "Symbiotic scheduling for virtual machines on SMT processors," in *2012 Second International Conference on Cloud and Green Computing*. IEEE, 2012, pp. 145–152.

[78] J. R. Bulpin and I. Pratt, "Hyper-threading aware process scheduling heuristics." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 399–402.

[79] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-bandwidth aware thread allocation in multicore SMT processors," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 123–132.

[80] A. Fedorova, M. Seltzer, and M. D. Smith, "A non-work-conserving operating system scheduler for SMT processors," in *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, vol. 33, 2006, pp. 10–17.

[81] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading." in *USENIX Annual Technical Conference*, 2016, pp. 309–322.

[82] S. Siddha, V. Pallipadi, and A. Mallick, "Chip multi processing aware linux kernel scheduler," in *Linux Symposium*. Citeseer, 2005, p. 193.

[83] M. Liebowitz, C. Kusek, and R. Spies, *VMware VSphere performance: designing CPU, memory, storage, and networking for performance-intensive workloads*. John Wiley & Sons, 2014.

[84] "KVM: Dynamic Halt Polling Patches," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=aca6ff29c4063a8d467cdee241e6b3bf7dc4a171.

[85] "Dynamic Halt Polling Technique," https://lkml.org/lkml/2017/6/22/296.

[86] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208464

[87] W. Jia, C. Wang, X. Chen, J. Shan, X. Shang, H. Cui, X. Ding, L. Cheng, F. C. M. Lau, Y. Wang, and Y. Wang, "Effectively mitigating i/o inactivity in vcpu scheduling," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018.

[88] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 329–341. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu_yunjing

[89] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 5.

[90] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang, "Perfiso: Performance isolation for commercial latency-sensitive services," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 519–532. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/iorgulescu

[91] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. IEEE, 1999, pp. 54–58.

[92] "Intel 64 and ia-32 architectures developer's manual," https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[93] C. Ruemmler and J. Wilkes, "UNIX disk access patterns," in *Winter USENIX Conference*, 1993, p. 405–420.

[94] D. Le Moal, "I/o latency optimization with polling," *Vault–Linux Storage and Filesystem*, 2017.

[95] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230.

[96] M. Ben-Yehuda, M. Factor, E. Rom, A. Traeger, E. Borovik, and B.-A. Yassour, "Adding advanced storage controller functionality via low-overhead virtualization." in *FAST*, vol. 12, 2012, pp. 15–15.

[97] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, no. 184, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1594371.1594375

[98] "L1 Terminal Fault," https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[99] "Flushing L1 Data Cache When a vCPU Enters the Guest OS," https://lore.kernel.org/patchwork/patch/974356/.

[100] “Flushing TLB When a vCPU Enters the Guest OS,” https://lwn.net/Articles/740363/.

[101] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, “Demand-based coordinated scheduling for smp vms,” in *ACM SIGPLAN Notices*, vol. 48. ACM, 2013, pp. 369–380.

[102] L. Vilanova, N. Amit, and Y. Etsion, “Using SMT to accelerate nested virtualization,” in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 750–761.

[103] A. Snavely, D. M. Tullsen, and G. Voelker, “Symbiotic jobscheduling with priorities for a simultaneous multithreading processor,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1. ACM, 2002, pp. 66–76.

[104] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, “A performance counter architecture for computing accurate cpi components,” *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, pp. 175–184, 2006.

[105] S. Eyerman and L. Eeckhout, “Per-thread cycle accounting in SMT processors,” *ACM Sigplan Notices*, vol. 44, no. 3, pp. 133–144, 2009.

[106] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, “Symbiotic job scheduling on the ibm power8,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 669–680.

[107] S. Eyerman and L. Eeckhout, “Probabilistic job symbiosis modeling for SMT processor scheduling,” in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 91–102.

[108] “The HALT-Polling Kernel Module,” https://patchwork.kernel.org/patch/11030651/, 2019.

[109] “The Halt Polling Technique,” https://lwn.net/Articles/384146/.

[110] “Redis In-memory Key-Value Database,” http://redis.io/.

[111] “MySQL Database,” http://www.mysql.com/, 2014.

[112] “Apache Web Server,” http://www.apache.org, 2012.

[113] “RocksDB NoSQL Storage System,” https://rocksdb.org/. [Online]. Available: https://rocksdb.org/

[114] “PostgreSQL DBMS Benchmarks,” https://www.postgresql.org, 2012.

[115] “Apache spark benchmarks,” https://spark.apache.org/examples.html.

[116] “TPC-W Database Benchmarks,” http://osdldbt.sourceforge.net/.

[117] “XGBOOST Runtime System,” http://dmlc.cs.washington.edu/xgboost.html.

[118] “Sockperf,” https://github.com/Mellanox/sockperf.

[119] 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*. ACM, 2012, pp. 3–14.

[120] J. Ahn, C. H. Park, T. Heo, and J. Huh, “Accelerating critical os services in virtualized systems with flexible micro-sliced cores,” in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys ’18. New York, NY, USA: ACM, 2018, pp. 29:1–29:14. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190521

[121] B. Teabe, A. Tchana, and D. Hagimont, “Application-specific quantum for multi-core platform scheduler,” in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys ’16, 2016, pp. 3:1–3:14.

[122] X. Song, J. Shi, H. Chen, and B. Zang, “Schedule processes, not VCPUs,” in *APSys 2013*, 2013, pp. 1:1–1:7.

[123] L. Cheng, J. Rao, and F. Lau, “vScale: automatic and efficient processor scaling for smp virtual machines,” in *EuroSys 2016*. ACM, 2016, p. 2.

[124] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, “Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 73–84.

[125] X. Song, H. Chen, B. Zang, X. SONG, H. CHEN, and B. ZANG, “Characterizing the performance and scalability of many-core applications on virtualized platforms,” *Parallel Processing Institute Technical Report Number: FDUPPITR-2010*, vol. 2, 2010.

[126] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, “Smite: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 406–418.

[127] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, “Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 15–27.

[128] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, “Addressing fairness in SMT multicores with a progress-aware scheduler,” in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 187–196.

[129] D. Koufaty and D. T. Marr, “Hyperthreading technology in the netburst microarchitecture,” *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[130] A. Herdrich, R. Illikkal, R. Iyer, R. Singhal, M. Merten, and M. Dixon, “SMT QoS: Hardware prototyping of thread-level performance differentiation mechanisms,” in *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2012.

[131] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, “Scheduling with implicit information in distributed systems,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 26. ACM, 1998, pp. 233–243.

[132] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, “The interaction of parallel and sequential workloads on a network of workstations,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 23. ACM, 1995, pp. 267–278.

[133] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, “Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms,” in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 126–136.

[134] C. Weng, Z. Wang, M. Li, and X. Lu, “The hybrid scheduling framework for virtual machine systems,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 111–120.

[135] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, “Dynamic coscheduling on workstation clusters,” in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1998, pp. 231–256.

[136] P. G. Sobalvarro and W. E. Weihl, “Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors,” in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 106–126.

[137] A. C. Arpaci-Dusseau, "Implicit coscheduling: coordinated scheduling with implicit information in distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 283–331, 2001.

[138] Drummonds, "Co-scheduling SMP VMs in VMware ESX server," 2008, http://communities.vmware.com/docs/DOC-4960.

[139] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling: Mitigating load imbalance and improving utilization of heterogeneous resources," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 10–pp.

[140] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective distributed scheduling of parallel workloads," *SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 25–36, May 1996. [Online]. Available: http://doi.acm.org/10.1145/233008.233020

[141] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs," in *ACM ASPLOS 2013*, 2013, pp. 369–380.

[142] J. Turner, "New directions in communications(or which way to the information age?)," *IEEE communications Magazine*, vol. 24, no. 10, pp. 8–15, 1986.

[143] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

[144] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[145] "Parallel BZIP2 (PBZIP2)," http://compression.ca/pbzip2/, 2011.

[146] "Stanford parallel applications for shared memory (SPLASH)," http://www-flash.stanford.edu/apps/SPLASH/, 2007.

[147] "The Princeton application repository for shared-memory computers (PARSEC)," http://parsec.cs.princeton.edu/, 2010.

[148] "SPLASH2X Benchmark Suite," http://parsec.cs.princeton.edu/parsec3-doc.htm.

[149] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *EuroSys 2011*. ACM, 2011, pp. 257–272.

[150] "The Source Codes of Demand-Based Coscheduling," https://github.com/hwanju/demand-cosched.

[151] A. C. Arpaci-Dusseau and D. E. Culler, "Extending proportional-share scheduling to a network of workstation." in *PDPTA*, 1997, pp. 1061–1070.

[152] K. Wang, Y. Wei, C.-Z. Xu, and J. Rao, "Self-boosted co-scheduling for smp virtual machines," in *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2015, pp. 154–163.

[153] "Xgboost: A scalable tree boosting system," http://dmlc.cs.washington.edu/xgboost.html.

[154] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and distributed Computing*, vol. 16, no. 4, pp. 306–318, 1992.

[155] D. Feitelson and L. Rudolph, "Distributed hierarchical control for parallel processing," *Computer*, vol. 23, no. 5, pp. 65–77, 1990.

[156] C. Weng, Q. Liu, L. Yu, and M. Li, "Dynamic adaptive scheduling for virtual machines," in *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 239–250.

[157] Y. Yu, Y. Wang, H. Guo, and X. He, "Hybrid co-scheduling optimizations for concurrent applications in virtualized environments," in *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*. IEEE, 2011, pp. 20–29.

[158] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical cpu scheduler for multimedia operating systems," in *OSDI*, vol. 96, 1996, pp. 107–121.

[159] J. Nieh, C. Vaill, and H. Zhong, "Virtual-time round-robin: An o (1) proportional share scheduler." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 245–259.

[160] C. A. Waldspurger and W. E. Weihl, *Stride scheduling: Deterministic proportional share resource management*. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.

[161] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real-rate scheduling," in *OSDI*, vol. 99, 1999, pp. 145–158.

[162] K. J. Duda and D. R. Cheriton, “Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler,” *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 261–276, 1999.

[163] G. W. Dunlap, “Scheduler development update,” *Xen Summit Asia*, 2009.

[164] L. Cherkasova, D. Gupta, and A. Vahdat, “Comparison of the three cpu schedulers in xen,” *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, Sep. 2007. [Online]. Available: http://doi.acm.org/10.1145/1330555.1330556

[165] D. Petrou, J. W. Milford, and G. A. Gibson, “Implementing lottery scheduling: Matching the specializations in traditional schedulers.” in *USENIX Annual Technical Conference, General Track*, 1999, pp. 1–14.

[166] C. A. Waldspurger and W. E. Weihl, “Lottery scheduling: Flexible proportional-share resource management,” in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI ’94. Berkeley, CA, USA: USENIX Association, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267638.1267639

[167] J. Rao and X. Zhou, “Towards fair and efficient smp virtual machine scheduling,” in *ACM SIGPLAN Notices*, vol. 49. ACM, 2014, pp. 273–286.

[168] Y. Zhao, K. Suo, X. Wu, J. Rao, S. Wu, and H. Jin, “Preemptive multi-queue fair queuing,” in *Proceedings of the 28th international symposium on High performance distributed computing*. ACM, 2019.

[169] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center.” in *NSDI*, vol. 11, 2011, pp. 22–22.

[170] C. Delimitrou and C. Kozyrakis, “Quasar: resource-efficient and qos-aware cluster management,” *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[171] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, “Cpi2: Cpu performance isolation for shared compute clusters,” in *EuroSys 2013*. ACM, 2013, pp. 379–391.

[172] B. Urgaonkar, P. Shenoy, and T. Roscoe, “Resource overbooking and application profiling in shared hosting platforms,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 239–254, 2002.

[173] N. Vasić, D. Novaković, S. Miucin, D. Kostić, and R. Bianchini, “Dejavu: accelerating resource allocation in virtualized environments,” in *ACM SIGARCH computer architecture news*, vol. 40. ACM, 2012, pp. 423–436.

[174] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in *Eurosys 2013*, 2013.

[175] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations,” in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 248–259.

[176] 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*. ACM, 2011, p. 47.

[177] C. Delimitrou, D. Sanchez, and C. Kozyrakis, “Tarcil: reconciling scheduling speed and quality in large shared clusters,” in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 97–110.