UNDERSTANDING AND OPTIMIZING PARALLEL PERFORMANCE IN MULTI-TENANT CLOUD

by
YONG ZHAO

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2019

UNDERSTANDING AND OPTIMIZING PARALLEL PERFORMANCE IN MULTI-TENANT CLOUD

The members of the Committee approve the doctoral dissertation of Yong Zhao

Jia Rao
Supervising Professor _____

Song Jiang _____

Hong Jiang _____

Jiang Ming _____

Dean of the Graduate School _____

ACKNOWLEDGEMENTS

As a part of life journey, the time point to finish my six years Ph.D. program is approaching. However, I still remembered the feelings of some panic and uncertainty for the future when arrived at Colorado Springs for the first time. I still remembered the scene when our first paper was finally accepted after it was rejected many times. I still remembered I always received the timely support and encouragement from my supervior, friends, parents and my wife when I met with difficulties, felt upset in the work and daily life.

First, I want to send my sincere thanks to my advisor Dr. Jia Rao who gave me a chance so that I could continue my Ph.D. study. Dr. Rao is an excellent group leader who not only clearly pointed out the hot research directions for us but also constantly provided bunches of insightful suggestions for our writing skills, presentation styles and technical details. For our research, I appreciated that Dr. Rao can continuously work together with us towards doing the impactful projects and publishing high quality publications. I still remembered the hard times Dr. Rao led us to discuss the new research ideas, talked the design, argued the details with each other and verified the design with the experiments step by step. I could not have these achievements without his help. Currently we are still on the way and never give up. Personally I feel ashamed since I always set up a big goal but seldom achieve it. For my life, Dr. Rao also provided me with lots of help such as the financial support and taught us how should we live and adapt into the American culture.

Second, I would like to give my thanks to all the committee members Dr. Song Jiang, Dr. Hong Jiang and Dr. Ming Jiang. I always obtained lots of insights when talked with Dr. Song Jiang who is a very interesting researcher and can present a big story from the details of small things. Dr. Hong Jiang and Dr. Ming Jiang are really nice professors who can respond immediately when you need help.

Third, I spent three years doing my research in the department of computer science at the University of Colorado Colorado Springs and University of Texas at Arlington respectively. Really thanks the department secretary and staff members who gave me help and patiently answered my questions. I am also very lucky working and playing with talented and funny guys in Colorado and Texas. Thanks Dr. Yanfei Guo, Dr. Dazhao Cheng, Dr. Rui Zhao who gave me invaluable suggestions when I started my Ph.D. journey as a beginner. Thanks Dr. Wei Chen, Dr. Kun Suo, Dr. Fan Ni, Dr. Xingbo Wu, Dr. Zhichao Yan who shared their research and technical experiences with me. Especially I want to mention Dr. Kun Suo who is always the first guy to embrace and play with the new things as well as has an accurate sense for their development. At the same time, I enjoyed the times talking

the crazy ideas with other system group members Xingsheng Zhao, Xiaofeng Wu, Hang Huang and Zhuo Huang.

Finally, I want to thank my parents for their unconditional support during my study. Although being the farmers with pretty low income and almost have no educational background, they have a long vision and worked really hard day and night to support the whole family. I would like to give special thanks to my wife, Miaomiao Li, for her love, understanding and support during happy and hard days.

August 1, 2019

ABSTRACT

UNDERSTANDING AND OPTIMIZING PARALLEL PERFORMANCE IN MULTI-TENANT CLOUD

Yong Zhao, Ph.D.
The University of Texas at Arlington, 2019

Supervising Professor: Jia Rao

As a critical component of resource management in multicore systems, fair schedulers in hypervisors and operating systems (OSes) must follow a simple invariant: guarantee that the computing resources such as CPU cycles are fairly allocated to each vCPU or thread. As simple as it may seem, we found this invariant is broken when parallel programs with blocking synchronization are colocated with CPU intensive programs in hypervisors such as Xen, KVM and OSes such as Linux CFS.

On the other hand, schedulers in virtualized environment usually reside in two different layers: one is in the hypervisor which aims to schedule vCPU onto each pCPU and another is in the virtual machine to schedule the processes. Such design principle will impose an implicit scheduling gap between these two layers such that threads holding the lock or waiting for the lock in the virtual machine can be inadvertently descheduled by hypervisors. This behavior will cause the well known LHP and LWP problems which can seriously degrade the performance of parallel applications.

While the cloud is believed to be an ideal platform for hosting parallel applications, its nature of multi-user sharing and resource over-commitment makes parallel performance often quite disappointing and unpredictable. Although many research works have identified the excessive synchronization delays such as LWP and LHP due to multi-tenant interferences as the culprit, there lacks a full understanding of the quantitative relationship between changes in synchronization and the overall performance loss. As performance modeling plays a fundamental role in designing traditional parallel systems, a systematic and quantitative study of parallel performance under cloud interferences would help improve the resource and power management in datacenters.

This dissertation explores two fundamental questions towards the solutions for the scheduling unfairness and inefficiency in multicore systems: why does the schedulers "unexpectedly" show to be unfairness under the common belief that scheduling algorithms

have been stable for many years and are already perfect? Why does the schedulers exhibit effectively regarding to scheduling the parallel applications in physical environment but perform badly in the cloud? The goal of this dissertation is to enable multicore systems to proactively anticipate and defend against the scheduling unfairness and inefficiency, rather than reacting to their manifestations and consequences.

This dissertation presents three key principles of systems design and implementation for rethinking and redesigning the scheduling algorithms in multicore systems against the unfairness and inefficiency—preemptive multiple queue fair queuing, interference-resilient scheduling, and differential scheduling. This dissertation demonstrates that applying these principles can effectively defend scheduling unfairness and inefficiency in multicore systems. Furthermore, this dissertation also presents the corresponding techniques and tools support that can automatically and systematically apply these principles into existing multicore systems.

Scheduling algorithms which originated from scheduling the packets in single-linked network were widely used in computer systems, however scheduling unfairness are unexpectedly manifested through scaling these algorithms from single core to multicore systems. Scheduling inefficiency are usually caused by the implicit semantic gap existing in the virtualized environment. Thus, this dissertation has modified the design of single-linked scheduling algorithm to make them to be fairness in face of the multiple-linked network and furthermore applied them into the multicore system scheduling to eliminate the unfairness. Instead of leaving the scheduling activities in two virtualized layers transparently for each other, this dissertation first characterized the performance of parallel applications under interference and then proposed methods to bridging the semantic gap in order to remove the scheduling inefficiency.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

Cloud computing, unlocked by virtualization technologies, is bringing a transformative change in enterprise infrastructures and shaping the way IT hardware is designed and deployed. Public cloud providers such as Microsoft Azure, Amazon AWS and Google Cloud has already provided the software-as-a-service, infrastructure-as-a-service and platform-as-a-service to the customers. For example, developers in the company with innovative ideas no longer require the large spending on the hardware to bring their service online or the human expense to manage it. They can lease virtual machines from cloud providers to deploy their service with a pay-as-you-go charging model. In order to fully utilize the hardware resource and reduce the energy cost, cloud providers preferred consolidating multiple independent workloads, each in a virtual machine, onto a fewer number of machines. As such, the success of cloud services critically depends on the effective management of datacenter resources. In this dissertation, we aim to understand the application performance in multi-tenant cloud to uncover the factors affecting the resource management, then propose the solutions to address the performance bottlenecks to improve the resource utilization.

In this chapter, we first introduce the motivation and background of this dissertation, then discuss the major challenges and present an overview of our solution.

1.1 Background and Motivation

Cloud Computing refers to the applications delivered as services over the Internet as well as the hardware and systems software in the datacenters that provide those services. By consolidating multiple independent workloads, each in a virtual machine, onto a fewer number of machines, cloud providers benefit from improved hardware utilization and significant energy savings. On the other hand, virtual servers can be configured according to the demand of cloud users. However, one difficulty for the cloud providers is whether they should consolidate more virtual machines on their hardware infrastructure to generate more revenue or avoid the loss of customer by providing the good performance for hosted applications. Performance guarantee has twofold meanings in a public cloud service: efficiency and fairness. First, cloud providers should deeply understand the characteristics of workloads in virtual machines such that datacenter resources can be managed to be better

utilized. At the same time, overhead caused by virtualization layer should be minimized to approximate the execution efficiency of applications in dedicated systems. Second, performance should be predictable and proportional to different applications, users and virtual machines.

In this dissertation, we focus on characterizing and understanding the performance of parallel applications in multi-tenant cloud. Based on these observations and insights, we propose the solutions to address these performance bottlenecks and also present a novel multicore scheduling algorithm to fairly allocate the hardware resources between different users, applications and VMs.

1.2 Challenges on Understanding and Optimizing the Parallel Performance

In this section, we mainly discuss the challenges on understanding and optimizing the performance of parallel applications under interference in multi-tenant cloud.

1.2.1 Scheduling Inefficiency

Scheduling efficiency especially in virtualized environment are constrained by the two implicit scheduling domains: (1) the guest OS schedules processes on vCPUs and (2) the hypervisor schedules vCPUs on physical CPUs. The scheduling activities in the guest OS are completely oblivious to the hypervisor and this behavior will cause several severe performance issues to the applications which employed synchronization primitives such as mutex lock. One well-known issue is the lock-holder preemption (LHP) [36] problem. LHP occurs when a vCPU is descheduled by the hypervisor while the thread currently running on that vCPU is holding an important lock. As the performance of parallel applications as a whole depends critically on the cooperation of multiple threads, if one thread holding the lock is preempted, other threads waiting for the lock are unable to make progress until the descheduled vCPU is rescheduled. Thus, the delay of one vCPU will significantly degrade the overall performance of the parallel program. Another issue is the Lock-waiter preemption (LWP). Instead of preempting the vCPU on which the thread are holding the lock, LWP happens when the hypervisor descheduled the vCPU on which the thread are waiting for the lock. As the lock will be passed to each thread according to their arriving sequence, a thread waiting for the lock is preempted will delay the acquiring of lock by other threads even the lock is released and free.

There have been studies [49, 55, 73, 85, 86, 110] narrowing the semantic gap by inferring scheduling events inside VMs at the hypervisor using heuristics, or approximating VM

coscheduling to mitigate the LHP problem, or allowing the guest OS to assist hypervisor scheduling. These approaches have their respective limitations. Different workloads require distinct heuristics to identify thread criticality; coscheduling is expensive to implement and causes CPU fragmentation; synchronization-oriented optimizations make the hypervisor scheduling very complex and can possibly compromise fairness between VMs.

1.2.2 Scheduling Unfairness

Classic scheduling problems revolve around setting the length of the scheduling quantum to provide interactive responsiveness while minimizing the context switch overhead to improve the scheduling efficiency, simultaneously accounting for the resources such as CPU, memory and I/O allocated to each user to keep the fairness, also catering to the batch and interactive workloads to maximize throughput, and efficiently managing the scheduler run queues to guarantee the load balance. By and large, by the year 2000, operating systems designers considered scheduling to be a solved problem. The Linus Torvalds said that you have to realize that there are not very many things that have aged as well as the scheduler, which is just another proof that scheduling is easy. His quote was an accurate reflection of the general opinion at that time.

However, this is not the case and a recent study [63] with the Linux scheduler revealed that the pressure to work around the challenging properties of modern hardware, such as non-uniform memory access latencies, high costs of cache coherency and synchronization, and diverging CPU and memory latencies, resulted in a scheduler with an incredibly complex implementation. In this work, four scheduler bugs are found and can cause the load imbalance in Linux such that some physical cores were left idle even there existed runnable threads waiting for their turn to run. These bugs undermine a crucial kernel sub-system, cause substantial, sometimes massive, degradation of performance. Another work [16] analyzed the scaling behavior of Linux kernel on a 48-core server with a set of applications that are designed for parallel execution and use kernel services. They found and provided a set of 16 scalability improvements to the Linux kernel.

Scheduling fairness, as a key algorithm in computer systems used to divide CPU cycles among users or VMs proportionally to their weights, was usually thought to be a solved problem. VMs, each hosting workloads independently, are usually consolidated into a single physical machine to improve the hardware utilization in virtualized environment. While fairness between different VMs is an important metric to measure the success of the public cloud providers such as Amazon EC2, Microsoft Azure and Google Compute engine platform, Rao etc. [87] found that the existing virtualized platforms fail to enforce fairness between VMs with different number of vCPUs that run on multiple CPUs. They attributed

the unfairness to the use of per-CPU schedulers and the load balance on these CPUs that incur inaccurate CPU allocations. Then, FlexW which aims to dynamically adjust the vCPU weights on multiple CPUs and FlexS which flexibly scheduled vCPUs to minimize wasted busy-waiting time were proposed to enhance fairness at VM-level. However, their method just fixed the symptoms instead of roots.

Actually, fair queuing (FQ) algorithms have been widely adopted in computer systems to share resources among multiple users. Modern operating systems and hypervisors such as CFS [68] in Linux and credit scheduler in Xen [118] use variants of FQ algorithms to implement the critical systems resource management. For example, Linux's CFS and Xen's credit scheduler are an implementation of the start time fair queueing (SFQ) [39] scheduling algorithm which originated from the Generalized Processor Sharing (GPS) [81] used for scheduling the networking packets. GPS allocated the capacity of a single-linked network to the competing flows in proportional to their weights (only the relative value of the weights are significant) if and only if all the flows are backlogged. However, GPS uses an idealized fluid model which cannot be realized in the real world. For integrated services networks (e.g., video and audio applications), start-time fair queueing [39] or SFQ showed it was better suited than WFQ [28] to provide fairness over servers with time varying capacity. SFQ associated each packet with start tag and finish tag and scheduled these packets in the increasing order of start tag.

1.3 Contributions

To fully unlock the potential of the guest OS in addressing the LHP and LWP problems, we design interference-resilient scheduling (IRS), a simple approach to bridging the guest-hypervisor semantic gap and guiding guest load balancing. Inspired by scheduler activations (SA) in hybrid threading, IRS notifies the guest OS and activates in-guest load balancing when a vCPU is to be preempted by the hypervisor. As such, lock holder threads can be promptly migrated to other running vCPUs to avoid LHP and LWP.

Symmetric Multiprocessing virtual machines (VMs) are becoming increasingly common in cloud datacenters and they are often used by cloud users to host parallel applications To fully utilize hardware parallelism, cloud providers prefer oversubscribing their datacenters by consolidating multiple independent VMs onto a single machine. The nature of multi-user sharing and resource over-commitment in the cloud often makes parallel performance quite disappointing and unpredictable. Although performance degradations caused by virtualization and interferences have been extensively studied, there still lacks a comprehensive understanding why parallel programs have unpredictable slowdowns when co-located with different types of workloads.

We presented a systematic and quantitative study of multithreaded performance under interference. We design synthetic workloads to emulate different types of interference and study the behavior of parallel programs under such interferences. We find that unpredictable performance is the result of complex interplays between the design of the program, the memory hierarchy of the host system, and the CPU scheduling at the hypervisor. To understand the intricate relationships between multiple factors, we decompose parallel runtime into compute, synchronization and steal time, and use the runtime breakdown to measure program progress and identify execution inefficiency under interference. Based on these findings, we develop an online approach to predicting performance slowdown without requiring parallel programs to be completed, and devise two scheduling optimizations at the hypervisor to reduce slowdowns.

While the existing FQ algorithms can enforce fair CPU allocation on a per-core basis, unfortunately there lacks an algorithm to fairly allocate CPU on multiple cores. As such, this common deficiency in state-of-the-art multicore schedulers causes unfair CPU allocations to parallel programs using blocking synchronization, leading to severe performance degradation. Parallel threads that frequently block due to synchronization exhibit deceptive idleness and are penalized by the thread scheduler. The discovered unfairness in multicore scheduling is the result of the complex interplay between parallel workloads and OS thread schedulers. On the one hand, parallel programs rely on simultaneous access to CPU to make collective progress among multiple threads and otherwise suffer substantial performance slowdown if critical threads holding important locks are preempted. The remaining threads who are waiting on the synchronization cannot make progress, either performing futile spinning or being put to sleep (block). On the other hand, multicore schedulers enforce fair CPU allocation on a per-core basis and are usually work-conserving. Therefore, threads that are idling due to synchronization forfeit their CPU shares, leading to unfair allocation between a parallel program and other competing programs. OS Load balancing could further aggravate this problem. Frequently idling threads, which show low CPU load, are gradually moved onto a few cores as consolidating fragmented load helps improve load balance. If CPU stacking occurs, sibling threads belonging to the same application compete with each other, introducing more idleness.

To address these issues, we extend the FQ algorithm for sharing a single network link to thread scheduling on multiple cores. We propose preemptive multi-queue fair queuing (P-MQFQ), a close approximation of the idealized generalized processor sharing (GPS) service discipline for multiple CPUs. P-MQFQ assumes a centralized queue to dispatch threads to multiple CPUs such that competing programs as a whole receive a fair share of the aggregated capacity of multiple CPUs. To tackle deceptive idleness, P-MQFQ allows threads from under- served programs to preempt currently running threads from other

programs. As such, programs experiencing deceptive idleness are temporarily prioritized to catch up with those who have exceeded their fair shares.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows.

Chapter 2 gives an overview on existing approaches on understanding and optimizing the performance and parallel applications. We start with the works characterizing the parallel performance in virtualized environment, then review the methods bridging the semantic gap between Guest OS and hypervisors and finally review the works focusing on improving the fair resource allocation in multicore systems.

Chapter 3 presents a systematic and quantitative study of multithreaded performance under interference. We design synthetic workloads to emulate different types of interference and study the behavior of parallel programs under such interferences. We find that unpredictable performance is the result of complex interplays between the design of the program, the memory hierarchy of the host system, and the CPU scheduling at the hypervisor. To understand the intricate relationships between multiple factors, we decompose parallel runtime into compute, synchronization and steal time, and use the runtime breakdown to measure program progress and identify execution in efficiency under interference. Based on these findings, we develop an online approach to predicting performance slowdown without requiring parallel programs to be completed, and devise two scheduling optimizations at the hypervisor to reduce slow-downs.

Chapter 4 finds a reverse semantic gap – the guest OS is oblivious of the scheduling events at the hypervisor, leaving the potential of addressing the LHP and LWP problems in the guestOS unexploited. Inspired by scheduler activations (SAs) in hybrid threading, we proposed interference-resilient scheduling (IRS), a guest-hypervisor coordinated approach to enhancing load balancing in the guest. IRS informs the guest OS before vCPU preemption happens at the hypervisor to activate in-guest load balancing. As such, critical threads on preempted vCPUs can be migrated to other running vCPUs so that the LHP and LWP problems are all alleviated.

Chapter 5 propose a preemptive multi-queue fair queuing (P-MQFQ) algorithm that uses a centralized queue to fairly dispatch threads from different programs based on their received CPU bandwidth from multiple cores. We demonstrate that P-MQFQ can be approximated by augmenting the existing load balancing in the OS without requiring to implement the centralized queue or undermining scalability. We implement P-MQFQ in Linux and Xen, respectively, and show significantly improved utilization and performance for parallel programs.

Chapter 6 concludes this dissertation with summaries of our approaches and directions for future work.

CHAPTER 2

Related Work

Deeply understanding the performance under interference can effectively aid the resource management in datacenters. As such, there are numerous works either modeling the resource contention on hardwares between different applications or proposing methods to reduce the synchronization delays and presenting new algorithms to address the fair resource allocation in shared environment. We will divide the relevant works into three parts and discuss them respectively.

2.1 Characterizing the Parallel Perfomance

Performance interference has been well studied in literature. Most work focused on contentions on shared resources, such as last-level caches [13, 20, 29, 41, 46], memory controllers [29, 64, 69], and hardware prefetchers [61], between sequential programs or multi-programmed/threaded workloads. There are also recent work measuring the interference in datacenters [25, 26, 51, 65, 74, 100, 121, 124]. These studies either assume space sharing between workloads [24, 83, 95, 96], use cycles per instruction (CPI) as a proxy of performance [25, 26, 121, 124] or use offline profiling to estimate the contentiousness of co-runners [65, 74]. These approaches can not be easily extended to manage performance interference of parallel applications. First, techniques addressing single-thread resource contentions do not necessarily optimize the execution of parallel programs as parallel performance is a function of single-thread computing and synchronizations. Second, widely used metrics such as CPI are not reliable in measuring performance in parallel programs because CPI can be either inflated or deflated due to synchronizations. Third, parallel applications are usually long-running jobs. Offline profiling is prohibitively expensive in production systems. In this work, we measure the amount of useful work completed to predict slowdown in an online manner.

There are also existing studies addressing the overhead of running parallel program in virtualized environments. Our optimizations are closely related to these works. Common issues include lock-holder preemption (LHP) [104], CPU stacking [98], and expensive traps to the hypervisor [32, 92]. Co-scheduling [76, 98] aims to schedule cooperative threads synchronously to avoid the LHP issue. However, our findings in this work show that *differ-*

ential scheduling may be more desirable for memory-bound programs. *Delayed preemption* shares the similar idea with demand-based coordinated scheduling [56] to temporarily delay the preemption of important threads. The scheduler in [56] delays the preemption of vCPUs that initiate wakeup IPIs to avoid LHP. Our purpose is to interleave the computations between parallel programs and interference to avoid future harmful preemptions. Ding et. al., found that consolidating multiple threads onto one vCPU to avoid blocking leads to significant performance boost in KVM [32]. Contrary to their findings, our experiments show that programs with fine-grained synchronization are more resilient to persistent interference in a Xen environment. The contradiction can be attributed to the different designs of Linux CFS scheduler and Xen's credit scheduler. To the best of our knowledge, this work is the first to study parallel performance under different types of interference and its complex interactions with scheduling on multiprocessors.

2.2 Interference Resilient Scheduling

Previous work attempting to eliminate this semantic gap can be divided into two categories: (1) Hypervisor-level approaches that treat the guest OS as a black box and (2) guest OS-assisted approaches employing para-virtualization. are dedicated to modifying the design of vanilla hyperviosr to make scheduling activities unknown to the Guest OS, and (2) Guest OS-assisted approaches employed para-virtualization techniques to customize Guest OS components separately.

2.2.1 Hypervisor Level Approaches

To deal with the LHP and LWP problems, VMware ESX 2.x [102] proposed the strict VM co-scheduling. This scheme allows vCPUs of the same SMP VMs to be synchronously scheduled and descheduled on different pCPUs. Despite its effectiveness in minimizing synchronization latency, it causes CPU fragmentation and vCPU priority inversion problems [93]. CPU fragmentation can lead to ineffective CPU utilization in environments where parallel applications are simultaneously hosted with sequential workloads [7, 34, 76, 91]. In order to lessen the severity of the CPU fragmentation problem caused by strict co-scheduling, relaxed co-scheduling [102] introduced in VMware ESX 3.x aimed to enable sibling vCPUs to make progress at similar paces and only requires the vCPUs that accrue enough skew to run simultaneously, while balance scheduling [99], a probabilistic co-scheduling scheme, increased the chance of co-scheduling sibling vCPUs by assigning them to different pCPUs. However, relaxed co-scheduling and balance scheduling distribute the sibling vCPUs across different pCPUs without considering the requirement for cooper-

ative scheduling between vCPUs of the same VM and these two approaches still have LHP and LWP problems.

Demand-based coordinated scheduling [55] adopted TLB shootdown IPI and reschedule IPI between different vCPUs as heuristics to identify cooperative vCPUs and proposed urgent vCPU first scheduling to prioritize vCPUs that are handling critical threads in the VM. Passive inference of guest OS events at the hypervisor is not applicable to many workloads. For example, IPI-based heuristics are not effective in identifying critical threads for parallel workloads with spinning synchronization. Although their proposed methods can reduce the occurrences of LHP problem, totally depending on the communication behaviors between vCPUs in the hypervisor will make their methods occasionally invalid. For example, If the threads of applications running in the Guest OS do not demonstrate this communication pattern (e.g., I/O intensive workloads), their methods are unable to retrieve the IPI used to do the vCPU scheduling and consequently lose its effectiveness. The root causes are ignoring the characteristics of applications and absent of the collaboration from Guest OS.

2.2.2 Guest OS-Assisted Approaches

To narrow the guest-hypervisor semantic gap, guest OSes are para-virtualized to coordinate with the hypervisor to avoid LHP and LWP. Dynamic adaptive scheduling [112] modified the guest OS to detect excessive spinning and report this information to the hypervisor. If a VM has reported frequent high spin waiting time, the hypervisor regards this VM as synchronization intensive and tries co-schedule its vCPUs as much as possible. regards OS-informed excessive wait time on spin-locks as an indicator to make vCPUs of a VM to be scheduled at the same time. It introduces a monitoring module in the Guest OS, this module periodically detects spin-locks with long wait time and reports this information to the hypervisor. However, this method has the security concerns as well as the CPU fairness problem. Uhlig et al., [105] proposed a delay preemption mechanism to minimize synchronization latency. Before a user-level thread acquires a spin-lock, the guest OS notifies the hypervisor of this pending event on the vCPU on which the thread is running. The notification requests that the vCPU not be preempted for a predefined period of time to avoid LHP and LWP.

The common issue of these guest OS-assisted approaches is that the guest OS is only responsible for passing down the information about lock holders and relies on the hypervisor to efficiently schedule vCPUs. As a result, the hypervisor needs to frequently deviate from its existing scheduling algorithm to utilize the semantic information for more efficient scheduling. Such invasive changes to hypervisor-level scheduling not only make

hypervisor design more complex but can compromise VM fairness. They also defer involuntary context switching of a lock-holder vCPU to minimize synchronization latency. In order to identify the lock-holder vCPU, they present an intrusive lock-holder preemption avoidance which lets the Guest OS notify the hypervisor not to preempt the vCPU for next n microseconds if the vCPU will immediately acquire the spin-lock. One of the challenges of this mechanism is to decide the numeric value of n which will be depended heavily upon the operating system and the characteristics of workload being run.

For non-intrusive scheme, the hypervisor monitors all the state switching of Guest OS between user-level and kernel-level based on the fact that the Guest OS will release all kernel locks before returning to user-level. As such, the hypervisor can determine whether it is safe to preempt the vCPUs without preempting the lock-holders.

2.3 Preemptive Multi-Queue Fair Queuing

Packet Scheduling. Fair queuing algorithms have been extensively studied in the literature. Most algorithms use the generalized processor sharing discipline [58] as the reference model. Each flow is assigned a weight and consists of requests or packets which are sent to the server or device sequentially. GPS will allocate the capacity of network link to the competing flows in proportional to their weights (only the relative values of the weights are significant) if and only if all the flows are backlogged. A flow is said to be backlogged at time point t if a positive amount of of that flows traffic is queued at time t. Since GPS uses an idealized fluid model and cannot be implemented in real systems, various packet-based approximations of GPS have been developed [11, 12, 14, 28, 37, 39, 40, 48, 70, 81, 82, 89, 122, 123]. Among these algorithms, weighted fair queuing (WFQ) [28] also known as Packet-by-Packet Generalized Processor Sharing (PGPS) [81] has been considered to be the best approximation to the GPS with respect to the accuracy. WFQ will schedule the packets in the increasing order of their departure times in the GPS. However, WF^2Q [12] demonstrates that WFQ can be far ahead of GPS in terms of number of bits served for a session which could cause large discrepancies between the services provided by WFQ and GPS. In order to mitigate this unfairness of WFQ, WF^2Q only consideres the packets which have already started receiving service in GPS system. For integrated services networks (e.g., video and audio applications), start-time fair queueing [39] or SFQ showed it was better suited than WFQ to provide fairness over servers with time varying capacity. SFQ associated each packet with start tag and finish tag and scheduled these packets in the increasing order of start tag. Furthermore, SFQ changed the definition of system virtual time $v(t)$ to be the start tag of the packet in the service time t. SFQ(D) [48] is adaption of start-time fair queueing (SFQ) for servers with a configurable

degree of internal concurrency and it makes it possible to share a server among multiple request flows. Hierarchical packet fair queueing [11] or H-PFQ proposed the WF^{2}Q+ algorithm which extended WF^{2}Q but with a lower complexity to provide the bounded-delay and fairness for hierarchical link sharing and traffic management between different service classes. All these fair sharing algorithms only provide methods for proportionally sharing a single server among competing flows. Unfortunately, they do not address the problem of fair sharing on multiple servers. MSFQ [14] extends WFQ to support multiple servers. It dispatches packets whenever there is an idle server and according to the order of packet completion under GPS. An important limitation of these packet-based fair queuing algorithms is that fairness is only guaranteed between backlogged flows and those that exhibit deceptive idleness are penalized under these algorithms.

CPU Fair Scheduling. Proportional share resource management provides a flexible and useful abstraction for multiplexing scarce resources such as the CPU, memory and disk among multiple users. The basic idea is to associate each user a weight and allocate resources to users in proportion to their weights. Many proportional fair share algorithms have been proposed to allocate CPU bandwidth [33, 35, 39, 43, 53, 60, 68, 71, 72, 80, 97, 108, 109].

On uniprocessor systems, early CPU schedulers [43, 53] were based on the concept of task priority and failed to precisely control the CPU allocation in proportion to user weights. and controlled the fair distribution of system resources such as memory, CPU to sets of related processes. Another fair share scheduler [53] was built on conventional priority scheduler and aimed to allocate CPU resource so that each user got their fair share over a long period. Lottery scheduling [108] is a randomized resource allocation mechanism based on the notion of a ticket. Compared to the traditional priority-based schedulers, lottery scheduling approximates proportional fair sharing in the long-term scheduling. implemented proportional-share resource management which guaranteed that the resource consumption rates of active computations were proportional to the relative shares that they were allocated. Lottery scheduling was especially desirable in systems that serviced requests of varying importance such as database, media-based applications and networks. Stride scheduling [109] further improves over lottery scheduling with significantly improved accuracy over relative throughput rates and less response time variability. The core allocation mechanism used by lottery and stride scheduling is based on WFQ or SFQ algorithms for packet scheduling. Hierarchical CPU scheduler [39] implemented the SFQ algorithm in multithreaded scheduling and could support hard and soft real-time, as well as best effort applications in a multimedia operating systems. Borrowed Virtual Time Scheduling [33] or BVT aimed to provide low-latency for real-time and interactive applications by introducing a latency parameter into SFQ algorithm. Specifically, BVT

scheduled the threads in the increasing order of the effective virtual time which was calculated by subtracting the value of latency parameter from start tag of each thread. Virtual time round robin [72] combined the round robin and WFQ fair queueing algorithm to guarantee proportional allocation CPU resource among a set of clients.

On multicore systems, Surplus Fair Scheduling [18] or SFS is derived from SFQ and demonstrates that GPS-based algorithms can provide strong fairness guarantees in uniprocessor environments, but they can result in unbounded unfairness when employed in multiprocessor environments. The reason is that GPS-based algorithms are unable to distinguish between feasible and infeasible weight assignments as well as are unable to achieve proportional allocation in the presence of frequent arrivals and departures. SFS measured the surplus service received by each thread and scheduled the threads in the increasing order of surplus value. The surplus service was the difference between the start tag of each thread and the system virtual time. SFS defined a global system virtual time which was the minimum value of start tag among all the runnable threads. Unlike the SFS, Deadline Fair Scheduling [19] sets the system virtual time to be the average value of start tags of all runnable threads. DFS calculates a counter value "deadline" for each thread and scheduled the threads in the increasing order of deadline. Hierarchical schedulers such as the Linux Completely Fair Scheduler [68] or CFS and Distributed Weighted Round Robin [60] extend fair queueing to multiple cores by maintaining per-core run queues and load balancing runnable threads across different cores. CFS implemented SFQ algorithm and maintained a separate virtual time for each core. As such, each core can make its scheduling decision independently. However, the drawback of this method is that CFS has to depend on an additional module in Linux to keep the load across difference cores balancing.

Virtualization technologies [1, 9, 54, 106] enable consolidating multiple independent workloads, each in a virtual machine, onto a fewer number of physical machines to improve the hardware utilization. For virtualized systems, such as public clouds (e.g., Amazon EC2 [4], Google Compute Engine [38] and Microsoft Azure [117]), fairness between tenants and efficiency of running their applications are the key to success. Many scheduling mechanisms have been developed [22, 50, 57, 75, 77, 84, 90, 111, 113, 116, 120, 125, 126] to improve the hardware resource utilization and allocation in virtualized environment. Flex [87] found that existing virtualization platforms failed to provide fairness between VMs and proposed FlexW which dynamically adjusted vCPU weights and FlexS which flexibly scheduled vCPU to minimize wasted busy-waiting time to enforce fairness. VMware [102] implemented a relaxed co-scheduling algorithm that is based on WFQ to achieve proportional fair allocation of CPU resources to different tenants. Gleaner [31]

employed resource retention and consolidation scheduling to combine short idle periods on multiple vCPUs into long idle periods to reduce harmful vCPU context switches in KVM.

CHAPTER 3

Characterizing and Optimizing Parallel Programs Under Interference

3.1 Introduction

Cloud computing, powered by warehouse-scale datacenters, provides users with abundant parallelism and potentially unlimited scalability. While the cloud is believed to be an ideal platform for hosting parallel applications, its nature of multi-user sharing and resource over-commitment makes parallel performance often quite disappointing and unpredictable. Although many studies [32, 56, 76, 98, 104, 114] have identified the excessive synchronization delays due to multi-tenant interferences as the culprit, there lacks a full understanding of the quantitative relationship between changes in synchronization and the overall performance loss. As performance modeling plays a fundamental role in designing traditional parallel systems, a systematic and quantitative study of parallel performance under cloud interferences would help improve the resource and power management in datacenters. Most importantly, predicting parallel performance allows users to evaluate the outcome of their cloud lease without completing long-running applications and select cloud services wisely to meet their expectations.

However, parallel performance is notoriously difficult to reason about in a shared cloud environment. Memory locality and resource contentions on the CPU function units, shared cache and memory bandwidth could all affect the speed of individual threads. More-

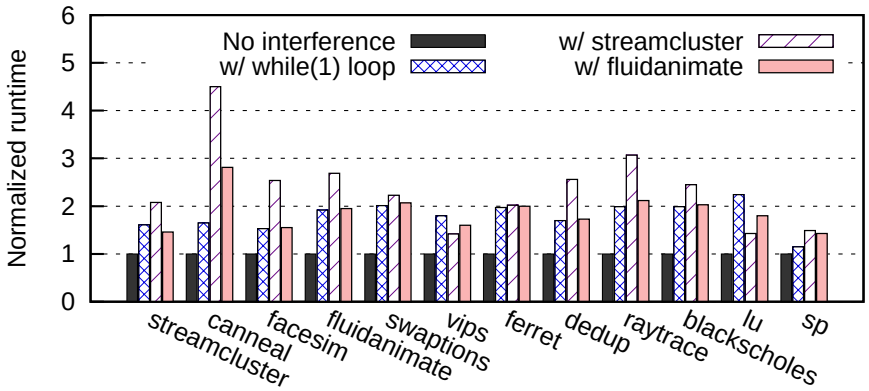


Figure 3.1: The runtime slowdown of parallel workloads under different interference scenarios.

over, the performance of parallel programs as a whole also depends critically on the cooperation of multiple threads. Contentions on CPU time at individual threads cause the well-known lock-holder preemption (LHP) problem [104], in which threads holding important locks are prematurely de-scheduled, leading to exceedingly long synchronizations and out-of-sync executions at multiple threads. The major challenges in modeling parallel performance are twofold: 1) individual threads have varying slowdowns when co-located with different interferences; 2) the quantitative relationship between slowdowns at individual threads and the overall slowdown of the parallel program remains unclear.

To demonstrate the severity of performance degradation and its unpredictability in multi-tenant clouds, Figure 5.1 shows the slowdown of various parallel programs under different interference scenarios. The parallel workloads include programs from the PARSEC [103] and NASA parallel benchmarks (NPB) [8] with different synchronization methods (i.e., busy-waiting and blocking), varying task granularities, and different work assignment policies (i.e., static and dynamic/work-stealing assignments). The interferences contain a synthetic workload that only competes for CPU cycles (i.e., `while(1)` loop) and two real workloads from the PARSEC benchmarks. Both the parallel workloads and the interferences run with 4 threads (see Section 3.3.1 for detailed settings). From the figure, we can see that performance slowdowns vary substantially across different workloads, even under the same type of interference. For example, the synthetic loop incurred more than 100% slowdown to *lu* while it merely slowed *sp* by 13%. More interestingly, slowdowns become more unpredictable when the background interference changes. For example, while *facesim* suffers a 165% slowdown with *streamcluster*, its degradation with *fluidanimate* is only 53%. In contrast, some workloads suffer more severe slowdowns under *fluidanimate* compared to that under *streamcluster*, e.g., *lu* and *vips*. The difference in memory contentiousness of the interfering workloads alone can not explain the varying slowdowns. The unpredictability is the result of complex interplays between the designs of the program, the memory hierarchy of the host machine, and the underlying CPU scheduling at the hypervisor. Understanding how these factors interact with each other is crucial to fully understanding parallel performance in the cloud.

In this chaper, we aim to uncover the mystery of parallel performance under interference. Our methodology is to study parallel performance in controlled experiments so that individual factors can be examined separately. To this end, we design a set of synthetic workloads to emulate different types of interference. The synthetic workloads include `while(1)` loops with *persistent*, *periodic* and *intermittent* interference patterns and configurable CPU demands. To further pinpoint the source of unpredictability, we design a profiling tool, `vProfile`, to derive a detailed breakdown of parallel runtime, and to report important scheduling and hardware statistics. With the help of fine-grained

profiling, we have the following findings: 1) the slowdown of individual threads depends on the amount of CPU allocated to the thread, which is largely affected by the synchronization granularity of the program in a shared environment; 2) while interference is believed to slow down individual threads due to contentions on shared resources, it can constructively accelerate threads in memory-bound applications due to the alleviation of intra-program contentions; 3) Uncoordinated scheduling of co-running parallel programs causes out-of-sync execution between multiple threads, leading to further performance degradation.

Based on these findings, we develop an online approach to predicting the performance of parallel programs under interference. Our online prediction first profiles parallel execution under interference for a short period of time and compares the completed *useful* work during sampling with that in a dedicated environment during the same length of period. The difference in completed useful work is used to predict the overall slowdown. We define useful work as the necessary computation needed by a thread assuming an ideal memory system with zero latency and perfect synchronization with no spinning and blocking cost. Specifically, we determine useful work by removing cycles that are stolen by other tenants, perform spinning or context switching (i.e., sleep and wakeup), and are spent in the memory hierarchy from the profiling sample. Experimental results show that, for regular parallel programs in which individual threads are assigned the same amount of work and perform iterated computations, the online approach achieves on average less than 4.5% errors in predicting the overall slowdown.

We also devise two scheduling optimizations at the hypervisor to reduce slowdowns. To avoid premature preemptions, we propose *delayed preemption* (DP) to interleave the computations of the parallel program and background interferences. As programs may afford different lengths of delay, we further make DP adaptive to the varying synchronization granularities for different programs. Experimental results show that DP improves the performance of PARSEC benchmarks by up to 23%. Another optimization motivated by our analysis is *differential scheduling* (DS), which purposely avoids co-scheduling of parallel threads by having different time slices on multiple CPUs. Results show that this simple technique outperforms stock Xen by as much as 38% in NPB benchmarks.

The rest of the chapter is organized as follows. Section 3.2 introduces the design of vProfile and the synthetic interferences. Section 3.3 provides an in-depth analysis of parallel performance under different types of interference. Section 3.4 and 3.5 present an approach for online performance prediction and two hypervisor-level optimizations motivated by the analysis, respectively. Section 3.6 discusses limitations and Section 4.6 concludes this chapter.

3.2 Profiling the Performance of Multithreaded Programs

The key to understanding parallel performance under interference is to identify the sources of slowdown. A breakdown of parallel runtime from experiments in a well-controlled environment would help pinpoint the culprit and inspire possible remedies. In general, parallel runtime consists of compute time and idle time. Traditionally, idle time due to load imbalance and synchronization is considered the major source of parallel overhead as no progress can be made during idle time. In a virtualized environment, guest operating systems (OSes) schedule application threads onto virtual CPUs (vCPUs) and multiple vCPUs from different virtual machines (VMs) can share the same physical CPU (pCPU). Steal time is the time a vCPU waits to run on a pCPU while the hypervisor is servicing another vCPU. A large steal time indicates severe contentions on the CPU allocation. Therefore, we decompose parallel runtime into *compute*, *synchronization* and *steal* time to study the causes of slowdown. We design `vProfile` to report the breakdown of parallel runtime and record important scheduling statistics in the hypervisor. vProfile provides two hypercalls [1]: `vprofile_start` and `vprofile_stop` to mark the start and end of a profiling period.

3.2.1 Decomposing Parallel Runtime

During profiling, vProfile tracks per-vCPU state changes in the hypervisor. Xen defines four vCPUs states, i.e., `running`, `runnable`, `blocked`, and `offline`. Steal time can be accounted using time spent in the runnable state, which counts the time a vCPU is ready to run but fails to acquire the pCPU. The accounting of synchronization time depends on the synchronization methods used by the parallel program. For blocking synchronization, such as `mutex` and `semaphore`, synchronization time is simply the time a vCPU stays in the blocked state.

Accounting synchronization time is more challenging for programs using busy-waiting synchronization (e.g., spinlocks) as vCPUs are always in the running state. There is existing work detecting spinning by instrumenting guest OS kernels [114], tracking user-kernel mode switches [104], and monitoring hardware performance events [17, 88]. We use the lightweight spin detection proposed in [88] to break the time in the running state into compute and synchronization (or spinning) time. As spin loops usually contain only a few instructions and are executed repeatedly, spinning vCPUs show high branch per instruction and low branch miss prediction rates compared to sibling vCPUs that are performing regular computation. We add a new `spinning` state to Xen and place a vCPU to such

[1] We describe the design of vProfile in a Xen environment. Other hypervisors can be easily modified to support vProfile.

a state when spinning is detected. Synchronization time is then the time a vCPU stays in the spinning state.

3.2.2 Recording Performance Events

The breakdown of execution time alone is not sufficient to identify the causes of slowdown. For example, an increase of synchronization could be due to long latencies at a few synchronization points or prolonged wait time at many places. Detailed execution statistics can help find the root cause and develop approaches to mitigate slowdowns. vProfile reports per-vCPU statistics of the scheduling and hardware performance events listed in Table 3.1.

Event	Description
`YIELD`	Voluntary yield to other vCPUs due to idling
`PREEMPT`	Involuntary preemption by the scheduler
`IDLE`	The time the pCPU in the idle state
`HARDWARE_STAT`	Statistics from hardware performance counters

Table 3.1: vProfile performance events.

Events `YIELD` and `PREEMPT` shed light on the contentions between co-located VMs while `IDLE` reflects the overall utilization of the pCPU as well as the efficiency of the scheduler. Performance statistics from hardware counters can trace the low-level program behaviors under interference and reveal the complex interactions between the program and the hardware. vProfile can be configured to track various hardware statistics, including cycles spent in the offcore memory system (`OFFCORE_STALL`), LLC misses per thousand instructions (`MPKI`), and other events related to cache coherence traffic between private L2 caches.

3.2.3 Enabling Dedicated Mode

vProfile enables online performance prediction by temporarily throttling co-located workloads to estimate their reference performance. It provides a short period of dedicated execution to emulate the performance on a dedicated machine. We assume that the sampling should cover a number of major iterations of the parallel program and contain sufficient information for performance prediction. The dedicated mode can be enabled multiple times once program phase change is detected. vProfile can integrate existing phase change detection techniques.

vProfile exports a new hypercall `sys_enable_dedicated` to the VM hosting the parallel application. Upon receiving the hypercall, the hypervisor takes vCPUs other than the calling VM's vCPUs off pCPUs' run queues and places them into the `offline` mode. The hypervisor freezes vCPUs of co-running VMs for a pre-defined period (e.g., 300 scheduling epochs). To minimize the impact of the dedicated mode on regular scheduling, we disable CPU time accounting (i.e., credit debiting in Xen's credit scheduler) during dedicated mode. As such, fair CPU allocation is not affected when normal execution is resumed.

3.2.4 Synthetic Interference

Synthetic workloads should slow down individual threads to faithfully reflect contentions on CPU time and shared resources, such as the last-level cache (LLC) and memory bandwidth. It should also be able to emulate the complex patterns of real workloads that simultaneously interfere with multiple threads. To this end, we design the synthetic workloads as simple CPU loops consisting of interleaved busy and idle intervals. The busy-to-idle ratio, which is configurable, determines the intensity of the interference. While the synthetic interference only contends for CPU cycles, it can emulate contentions on shared hardware resources because a decrease in allocated CPU time is equivalent to an increase in memory access cost for the parallel applications under test. We create the following three types of interference to study the complex interplay between the parallel program, the memory hierarchy and the underlying CPU scheduling:

- *Persistent* interference comprises of simple `while(1)` loops demanding 100% of CPU time. It emulates the CPU demand of long-running sequential jobs or parallel applications with busy-waiting synchronization. Due to its simplicity, it is scheduled by hypervisors at predictable time points and does not incur preemptions to the parallel threads.
- *Periodic* interference demands CPU at regular intervals or otherwise stays idle. The ratio of the CPU burst and the idle period determines the level of contention (i.e., CPU demand). Periodic interference has fixed burst-to-idle ratio and fixed length of computation at each interval. It emulates regular parallel applications that have predictable computations and synchronizations. Periodic interference is more complex than persistent interference as it sleeps and wakes up periodically, leading to preemptions of parallel threads.
- *Intermittent* interference demands CPU at irregular intervals. The ratio of CPU burst and sleep remains unchanged, but the length of computation changes randomly. It emulates multi-programmed workloads with independent (random) demands from

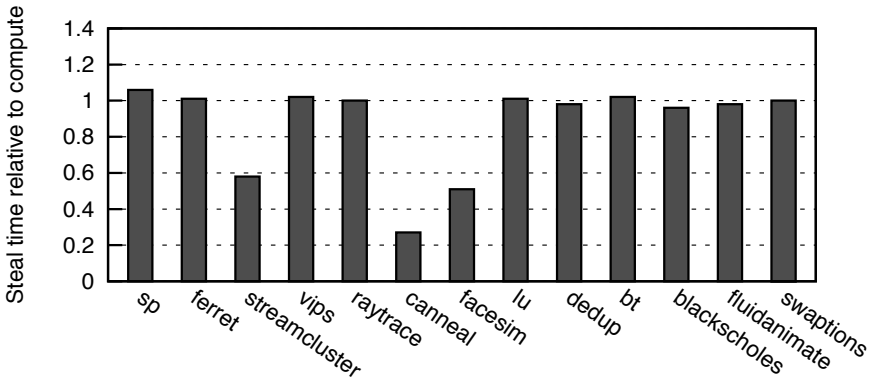


Figure 3.2: Varying resilience to interference.

individual threads or parallel applications with irregular CPU demands. Compared to periodic interference, whose computation and idling are predictable, intermittent interference has unpredictable demands. This helps to study the behavior of parallel programs when their execution is out-of-sync.

We use the method of differential analysis [62, 67] to compare the execution profiles of parallel programs under different types of interference. The low-level metrics that are highly correlated to the overall performance are examined to identify the causes of performance slowdown. We set the CPU demand of periodic and intermittent interference to 50% of the pCPU. The periodic interference performs 10ms computation and stays idle for 10ms. In contrast, the computation in intermittent interference varies randomly from 1ms to 40ms with the idle period changing accordingly to generate the 50% CPU demand. We use single-threaded interferences to study the slowdown of individual threads and the multi-threaded version to evaluate the efficiency of multiple parallel threads.

3.3 Understanding Parallel Performance

In this section, we use the statistics reported by vProfile to explain the mystery of parallel performance under interferences. We found that programs show different levels of resilience to interferences, leading to varying CPU allocations to the parallel program (Section 3.3.2). Interference may accelerate program execution by reducing the compute time needed in normal execution (Section 3.3.3). Finally, the overall performance is determined by the complex interplay between multiple factors (Section 3.3.4).

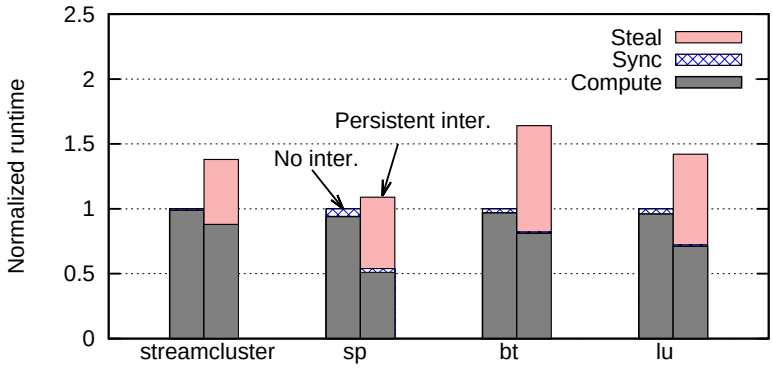


Figure 3.3: Interference reduces compute time.

3.3.1 Experimental Setup

We ran shared-memory multi-threaded programs on an Intel multicore machine with three types of interference. The host machine has a NUMA architecture with 16GB memory and two Intel Xeon E5620 2.40GHz 4-core processors. Each core in the processor has a private 256KB L2 cache and shares a 12MB L3 cache. Hyperthreading was disabled for all experiments. To isolate program performance from other factors, e.g., memory locality, we created the VM hosting the parallel programs in one memory node and pinned vCPUs of the VM to the processor affiliated with the node. Thus, all memory accesses were local and parallel threads shared the same last-level cache. We configured the VM running parallel programs with 4GB memory and 4 vCPUs, each pinned to a separate core in the same memory node. The background interfering VM had an identical configuration with all its vCPUs pinned to the same set of cores. Note that pinning vCPUs to pCPUs is to obtain reproducible results. We observed on average 55% performance slowdown and 15% variation when CPU affinity was turned off. We assigned equal weights to both VMs, assuming a fair allocation of the CPU.

We implemented vProfile in Xen 4.0.2 and modified Linux guest kernel 2.6.32 to use the profiling hypercalls. We selected the benchmarks in PARSEC 2.1 and the NAS Parallel Benchmark suite. The PARSEC benchmarks were compiled with *gcc-pthreads* and blocking synchronizations. We used the OpenMP version of the NPB benchmarks and set the environment variable `OMP_WAIT_POLICY` to active to use busy-waiting synchronization. All benchmarks were configured with 4 threads. We set vProfile to report performance statistics for the entire execution after programs complete.

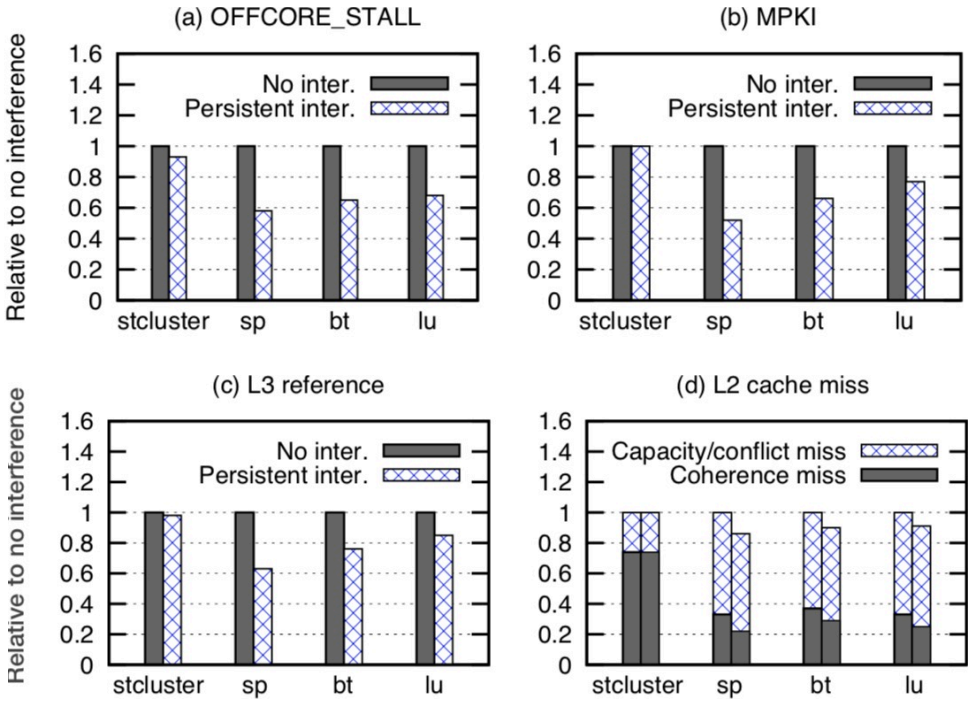


Figure 3.4: Interference alleviates intra-program contentions.

3.3.2 Varying Resilience to Interference

First, we study the slowdown at individual threads due to contentions on the CPU time. We placed a single-threaded persistent interference with one parallel thread and measured how much time was stolen (i.e., steal time) from the thread by the persistent interference. In this simple scenario, the thread co-located with the persistent loop would be the slowest thread in the parallel program, thereby deciding the overall performance. Figure 3.2 shows the steal time of various benchmarks relative to their compute time under the 1-loop persistent interference. Intuitively, the ratio of steal and compute time should be 1 if CPU is fairly allocated to the parallel program and the interference. However, Figure 3.2 suggests that some programs (i.e., *canneal*, *streamcluster* and *facesim*) be more resilient to interference and were stolen less time. An examination of program code revealed these benchmarks have fine-grained synchronizations and block frequently at synchronizing barriers. Zhou *et al.*, also showed that deliberately designed attacks can obtain excessive CPU allocations by exploiting the accounting vulnerabilities in Xen [127]. Harris *et al.* found the CPU time that each job receives can be drastically different and hard to control when multiple jobs run together [42]. In our experiments, the varying CPU allocation is due to the prioritization of latency-sensitive workloads in Xen.

Xen's credit scheduler considers vCPUs that wake up from sleep as latency-sensitive and assigns them a higher priority (i.e., the *boost* priority). Such vCPUs will preempt the current running vCPUs. Although the prioritization mechanism in theory benefits any programs that block, the granularity of synchronization plays an important role in gaining more CPU allocations. Xen implements a coarse-grained CPU scheduler which checks if the current running vCPU should be de-scheduled every 30ms (i.e., the default time slice). Whenever a vCPU is prioritized, it gains a full time slice unless voluntarily giving up the CPU, e.g., blocking due to synchronization. Thus, programs with fine-grained synchronization, e.g., those with computation less than 30ms between synchronization, are never forcibly de-scheduled by Xen due to the expiration of time slices, thereby being resilient to CPU contention. This issue is not specific to Xen and has also been observed in KVM [54].

As parallel programs have varying CPU allocations in response to interference, it is important to monitor *steal* time to determine the slowdowns at individual threads. However, the resilience to interference alone does not explain the unexpected marginal slowdown of *sp* in Figure 5.1, though it suffers significant steal in Figure 3.2. We uncover the reasons in the next subsection.

3.3.3 Varying Compute Time under Interference

In addition to the varying steal time, we find another factor that affects the slowdown at individual threads – the cost to access the memory hierarchy can change due to interference, leading to varying compute time. Figure 3.3 shows the runtime breakdown for programs that have unexpected slowdowns. These programs are expected to have slowdowns as much as their stolen time. From the figure, we can see that the computations needed to complete these programs decrease under the 1-loop persistent interference. The compute times for *streamcluster*, *sp*, *bt*, and *lu* in the complete program execution drop by 12%, 49%, 19%, and 29%, respectively.

To find the reasons for the reduced compute time, we show the statistics of hardware performance counters in dedicated execution and execution with 1 persistent interference in Figure 3.4 (a) - (d). The figures only show the statistics of the thread with interference and data is normalized to the no interference case (left bar in each group). `OFFCORE_STALL` refers to the cycles spent in the offcore memory subsystem on Intel processors. It measures the overall cost of accessing LLC and DRAM. `MPKI` measures LLC misses per thousand instructions. Figure 3.4 (a) suggests that these programs had fewer `OFFCORE` stalls under interference because on average there were fewer threads running simultaneously, thereby incurring less contention on the memory hierarchy. Figure 3.4 (b) shows that `MPKI` also drops under interference. Since the straggling thread progressed slower than sibling threads,

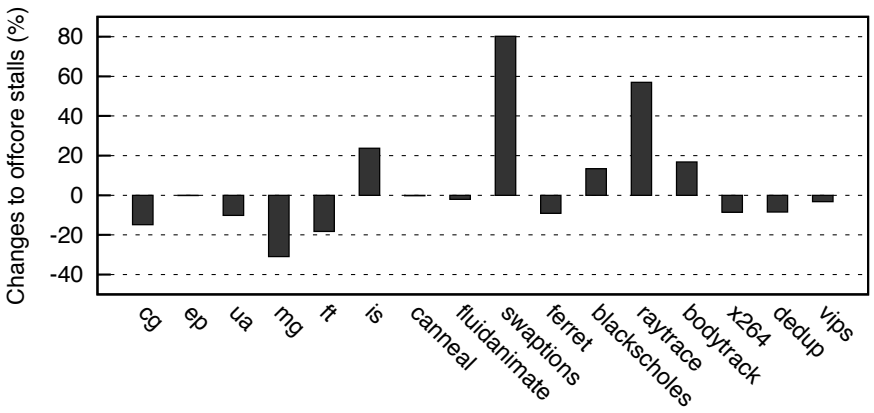


Figure 3.5: Interference changes memory access time.

it spent little time spinning and its instruction count decreased. A drop in the overall `MPKI` then indicates a significant drop in the number of LLC misses. Intuitively, the total LLC footprint of the parallel threads should remain unchanged, thus the LLC miss rate should not change.

We find that interference changes the way parallel applications interact with the memory hierarchy. Figure 3.4 (c) shows that the reduction in LLC misses was mainly due to fewer LLC references. For example, the number of LLC references for *streamcluster*, *sp*, *bt*, *lu* drop by 2%, 37%, 24%, 15%, respectively. Since LLC (L3 in our testbed) reference is the result of L2 misses, we further draw the breakdown of L2 misses in Figure 3.4 (d) to pinpoint the culprits of reduced L3 reference. Figure 3.4 (d) only shows demand data and instruction misses. Prefetching misses are excluded from the figure. We used L2 events `DEMAND.I_STATE` and `RFO.I_STATE` to count *coherence* misses and the remaining misses were *capacity* or *conflict* misses. The data shown in Figure 3.4 (c) and (d) is normalized to the no interference case.

From Figure 3.4 (d), we can see that the change in L2 coherence miss contributes most to the overall L2 miss reduction. A study of program source code reveals that most coherence misses are caused by *false sharing* between threads. The out-of-sync execution of parallel threads due to interference can constructively alleviate the contention on shared cache lines. Threads with different progression will see fewer cache line invalidations from sibling cores. Interference not only reduces the inter-core cache coherence traffic but also avoids some unnecessary L3 references. For example, miss on an unmodified L2 cacheline can be serviced by forwarding the line from a sibling core without sending requests to the L3 cache. This effectively reduces the number of capacity/conflict misses on the L3 cache. An exception is *streamcluster* whose change in compute time is mostly attributed to prefetching misses.

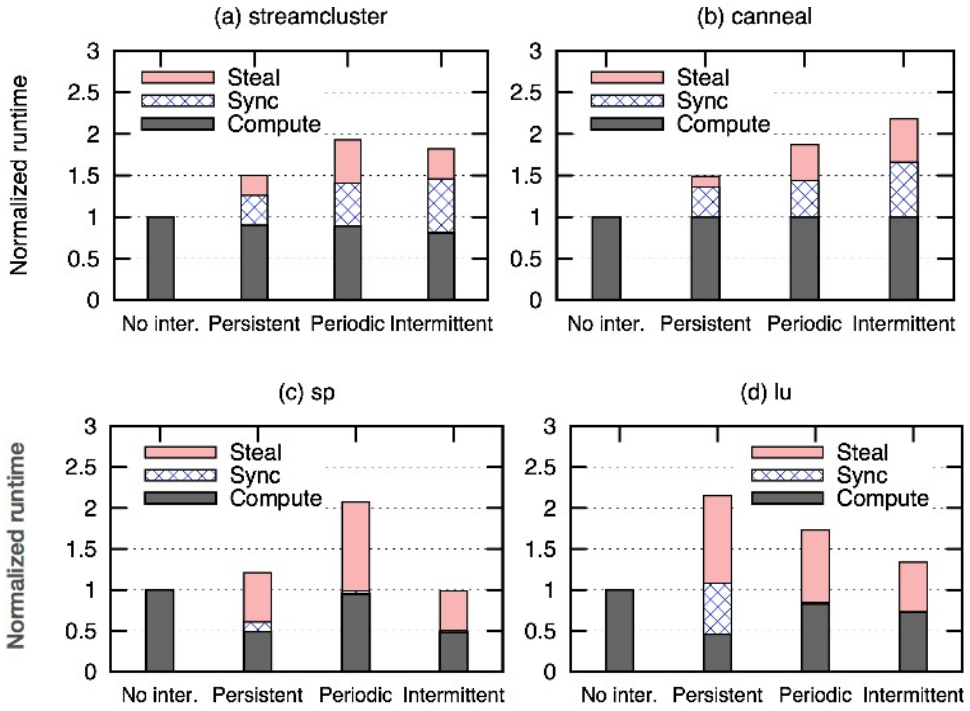


Figure 3.6: Program performance under different types of interference (4-loop).

Next, we show that most programs in the NBP and PARSEC benchmarks have varying memory cost under interference. Figure 3.5 shows the changes in `OFFCORE_STALL` under interference relative to the no interference case for the benchmarks not shown in Figure 3.4. Negative values suggest reductions in memory cost and vice versa. As we can see, the cost to access memory varies wildly from -30.9% to 80.2%, with most programs susceptible to such changes. If the compute time comprises mostly memory access time (i.e., memory-bound), the changes in memory cost due to interference could greatly affect the overall performance. While the increase in memory access time can be attributed to the loss of locality, e.g., threads dynamically stealing work from straggling threads in *raytrace* and *bodytrack*, the reduction in memory cost due to the mitigation of intra-program contentions can be exploited to improve datacenter efficiency. We carefully co-located *sp* with four intermittent interferences, with 5%, 8%, 10%, 15% CPU demands, respectively. The setting is to artificially create out-of-sync execution at parallel threads to mitigate the false sharing between threads.

Table 3.2 shows that *sp* achieved 9.0% better performance with 9.2% less allocated resource. This case study **suggests** a group of symbiotic datacenter workloads, in which seemingly destructive competitions help constructively mitigate intra-program contentions on shared resources.

	No inter.	w/ inter.
CPU %	400%	363% (**-9.2%**)
Runtime	1004s	914s (**+9.0%**)

Table 3.2: Improving *sp* performance with less resources.

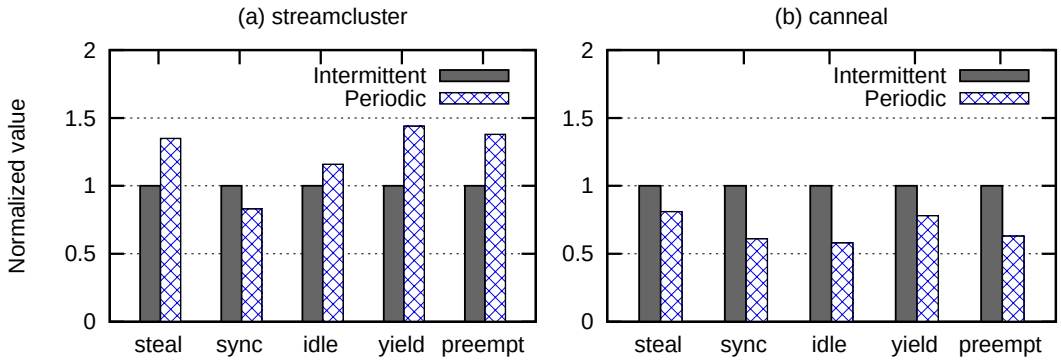


Figure 3.7: Low-level metrics under interference.

3.3.4 Complex Interactions with the Scheduler

Performance is even harder to characterize if all threads of the parallel program are affected by interference. In the simple one persistent interference scenario, program resilience to CPU contention or the change in compute time plays a major role in performance. In contrast, when all vCPUs have interference, the overall performance is a function of compute, sync, and steal time. Figure 3.6 shows the performance of parallel programs under different types of 4-loop interference, i.e., *persistent*, *periodic*, and *intermittent*. As shown in Figure 3.6(a) and (b), both *streamcluster* and *canneal* achieved the best performance under persistent interference but suffered under the other two. However, these two benchmarks behaved differently under periodic and intermittent interferences. While *streamcluster* had the largest slowdown under the periodic interference, *canneal* suffered most under the intermittent interference. Such uncertainty is due to the complex interactions among the parallel program, the interference, and the scheduler. For example, as discussed in Section 3.3.2, steal time is affected by the granularity of synchronization. A drop in compute time due to out-of-sync execution will decrease steal time but lead to increased sync time. The overall slowdown is determined by the interplays between these factors.

Figure 3.7(a) and (b) show the low-level performance metrics of *streamcluster* and *canneal* when co-running with periodic and intermittent interferences, respectively. The figures summarize the statistics for all vCPUs in the parallel VM. We find that low-level metrics shed light on the unpredictability of high-level performance. Except sync time,

all other low-level metrics are highly correlated with the overall slowdown. Among these metrics, *idle* time is the key to understanding the performance difference. `Idle` refers to the time neither the parallel program nor the interference was running. A longer idle time indicates a larger overlap between parallel program's computation and the CPU burst of the interfering loop. This results in more severe contentions on the pCPU. From the figure, we can see that longer idle time always leads to more preemptions of the parallel program and more yields, which is the sign of vCPU blocking due to imbalanced execution. The **implications** from these observations are that reducing the number of preemptions would help improve performance under interference and the system idle time is a good indicator of scheduling efficiency.

For programs with busy-waiting synchronization, such as NPB benchmarks *sp* and *lu*, both spin (or sync) time and compute time are counted as the CPU consumption of the parallel VM. Since steal time depends on the CPU usage of the VM, the performance of *sp* and *lu* is determined by their combined spin and compute time. Figure 3.6 (c) and (d) show the runtime breakdown of *sp* and *lu*, respectively. We make three key observations. First, co-running with intermittent interference achieved the best performance among the three interference scenarios. Second, persistent interference caused longer spin time than the other two. Third, periodic interference did not significantly increase sync time. These observations provide valuable insight into the complex interplays between busy-waiting workloads and interference.

The asynchrony in scheduling multiple vCPUs contributed most to the reduction of compute time under persistent and intermittent interferences. When co-locating the parallel program with persistent interference, individual vCPUs are likely scheduled in an uncoordinated manner on multiple pCPUs and the asynchrony remains until the completion of the parallel program because the scheduling rhythm (i.e., switching vCPUs at time slice expirations) on multiple pCPUs will not change for compute-bound workloads (i.e., spinning workload and persistent interference). However, the constant asynchrony continuously incurs exceeding spinning at faster vCPUs and eventually degrades overall performance. The randomness in intermittent interference also creates asynchrony on multiple vCPUs and helps reduce memory access cost. Contrary to persistent interference, it does not cause long spin time and neither does the periodic interference. The frequent switching between computation and idling (i.e., 10ms in periodic and 1-30ms in intermittent interferences) in these two interferences forces the hypervisor to perform scheduling at a much finer granularity compared to the default 30ms time slice in Xen. There have been studies using small time slices to improve the performance of virtualized IO [2, 119]. In our case, the fine-grained scheduling helps stop spinning vCPUs in a timely manner, which not only reduces the overall sync time but also saves the precious CPU time for useful work in the parallel

program. Our analysis motivates a possible **optimization** at the hypervisor to improve the performance of busy-waiting workloads: *differentiating scheduling on multiprocessors*.

3.4 Online Performance Prediction

In this section, we present an approach for online performance prediction based on the breakdown of parallel runtime. The idea is to sample parallel execution under interference and compare the execution profile with that in an interference-free environment. The key is to compare the amount of *useful* work completed in the two profiles and infer the slowdown from the difference in the speed of program progression. *We define useful work as the necessary work needed to complete the parallel job assuming an ideal memory system with zero latency and perfect load balancing*. As we have shown, the cost of accessing memory and performing synchronization can vary under interference, leading to dynamic computations required to complete parallel programs. In a machine with an ideal memory system, computation is performed in the CPU front-end which is an invariant in the presence of interference or unpredictable thread scheduling. With perfect load balancing, the time spent in synchronization, i.e., spinning or performing context switches, is almost zero. Thus, the compute time in such an ideal platform is only determined by compilation and the dynamic instruction scheduling on individual CPUs. As these two factors are not affected by interference, the ideal compute time is a **reliable metric** to measure program progress.

To measure useful work, or the compute time on an ideal platform, we remove the time spent in the memory hierarchy, synchronization, and the time stolen by other users, from the total time. Specifically, we calculate useful work t_{ideal} in a sampling period as follows:

$$t_{ideal} = t_{total} - t_{steal} - t_{sync} - t_{mem},$$

where t_{steal} measures the resilience of the program to interference and is reported directly by vProfile. We use the time spent on Intel's offcore memory subsystem (i.e., `OFFCORE_STALL`) to approximate t_{mem}. For programs with busy-waiting synchronization, t_{sync} refers to the spinning time recorded by vProfile. For programs with blocking synchronization, t_{sync} includes the time in the blocked state and the time performing vCPU context switches. While the blocked time is already included in the execution profile, we infer the cost of context switching by comparing the cycles spent by different threads in the same parallel program. Specifically, after removing t_{steal} and t_{mem} from the total time, we compare the remaining time of individual threads and attribute the difference to the varying numbers of context switches performed by them, assuming that each thread has been assigned an equal

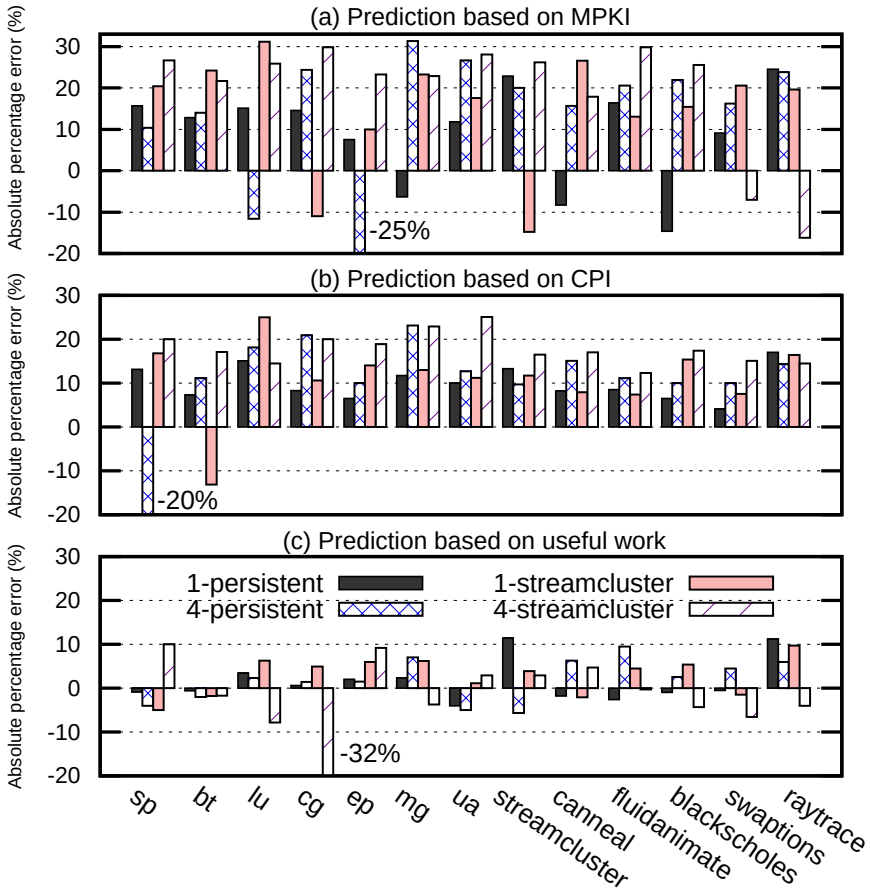


Figure 3.8: The accuracy of performance prediction.

amount of work. As such, we calculate the per context switch cost (in CPU cycles) and multiply the number of vCPU blocking to derive t_{sync}.

The online sampling profiles parallel execution in two steps. First, it enables the *dedicated mode* to collect statistics for an interference free execution. The dedicated mode sampling is then followed by a normal sampling with interference turned on. The length of the sampling can be tuned to produce the best accuracy. We empirically set the sampling length to 30 seconds. vProfile reports the runtime breakdown of all vCPUs in the VM under the two execution modes, respectively. The overall slowdown is then calculated as $\frac{t'_{ideal}}{t_{ideal}}$, where t'_{ideal} and t_{ideal} are the amount of useful work done by all threads in the parallel program during the dedicated mode and under interference, respectively.

Figure 3.8 shows the accuracy of the proposed online prediction compared with two representative prediction approaches based on misses per thousand instructions (MPKI) and cycles per instruction (CPI). We employed the same sampling-based method to predict the overall program slowdown using these two metrics. For example, the slowdown is predicted

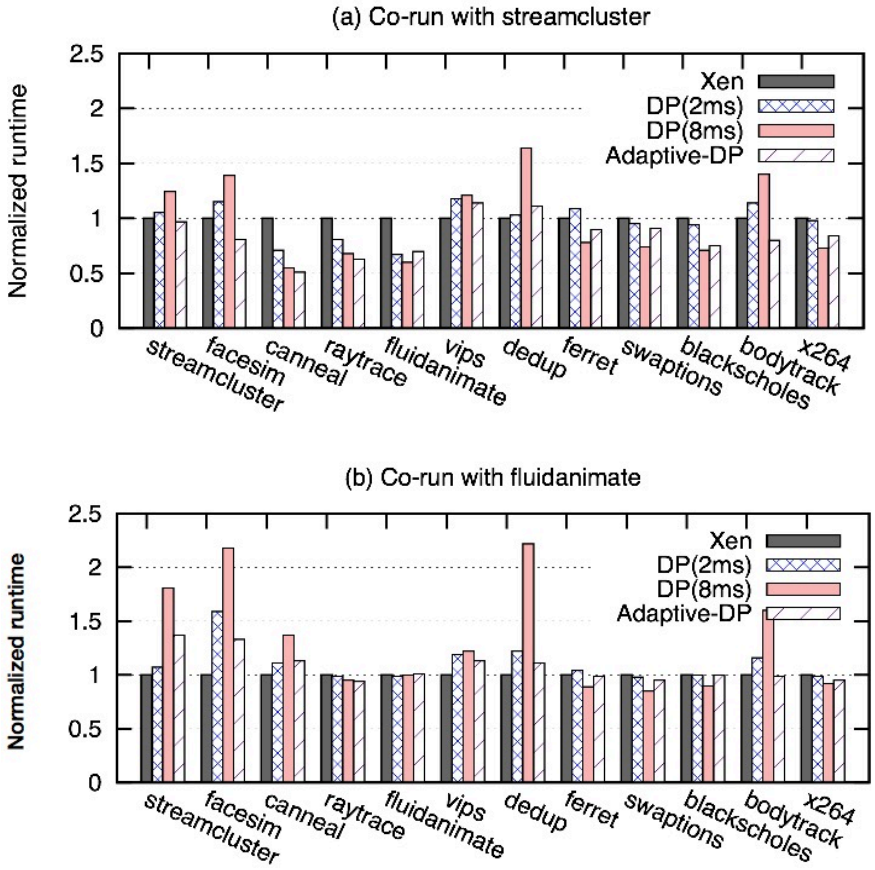


Figure 3.9: Performance of PARSEC benchmarks under *delayed preemption* (DP).

as $\frac{CPI}{CPI'}$ when using CPI to measure program progress under interference and under the dedicated mode. We used the synthetic workload and the real *streamcluster* workload as the interferences, respectively. For each type of interference, we evaluated the prediction with both the single-threaded (i.e., *1-persistent* and *1-stcluster*) and the multi-threaded (i.e., *4-persistent* and *4-stcluster*) interferences. With the synthetic persistent interference, which has predictable CPU demands and almost zero memory footprint, we test how well the prediction deals with the uncertainties due to parallel programs' varying resilience and memory accessing cost in response to interference. Then, we evaluate how accurate the prediction would be with real workload *streamcluster* that has varying CPU demands and contends on shared memory resources.

As shown in Figure 3.8 (a) and (b), MPKI and CPI-based predictions incurred significant prediction errors with on average 20.3% and 15.2% mean absolute percentage errors (MAPE), respectively, across all workloads. The inaccuracies were due to the misrepresentation of parallel performance by these two metrics. As shown in Figure 3.5,

memory access cost can either increase or decrease under interference but the overall performance is determined by multiple factors. Thus, memory-related metrics alone, e.g., MPKI, fail to accurately predict performance slowdown. CPI is a effective metric to predict the performance of serial programs as it measures the cost to execute instructions, assuming that the number of instructions needed to complete a program is constant. However, this assumption does not hold in parallel programs. The number of instructions executed by individual threads is variable due to the spinning and blocking performed by threads. For example, it is possible that a spinning thread making no execution progress can have a decreasing CPI because waiting on spinlocks does not cause any memory accesses and has a low CPI. Therefore, CPI alone does not capture synchronization in parallel programs and is not accurate in predicting the overall performance.

In contrast, as shown in Figure 3.8 (c), the performance predictions based on the useful work were accurate with an average MAPE of 4.5% across all workloads. In general, predictions with the synthetic interference are more accurate than that with *streamcluster* and predictions with single-threaded interference tend to incur less error. Except for *cg*, all predictions caused less than 10% errors even for *raytrace*, which implements dynamic work assignment at the user level to improve load balancing. While dynamic work assignment at the application level mitigates interference by assigning less work to straggling threads, it presents challenges to online performance prediction as the work done by individual threads does not reflect the overall progress. Our online prediction addressed this issue by counting useful work on all threads to measure program progress as a whole and achieved less than 10% prediction error for *raytrace*.

Sources of inaccuracy The accuracy of the prediction relies on one assumption: the sampling is representative of the overall parallel execution. This assumption do not always hold, leading to inaccurate predictions. For example, some applications, e.g., *cg*, have quite dynamic memory footprint at different execution stages. The dynamism can affect the effectiveness of the sampling-based prediction, especially when the interference is also dynamic. As shown in Figure 3.8, the prediction incurred 32% error on *cg* under the 4-thread *streamcluster* interference. As discussed in Section 3.3.2, the CPU allocation to *streamcluster* depends on its synchronization granularity. As *cg*'s memory demand changed, *streamcluster* had varying computation between synchronizations, thereby affecting the CPU allocation to *cg*. Thus, predictions based on one sample of the execution of such applications will be likely inaccurate.

3.5 Optimizations

Our analysis in Section 3.3 found that involuntary preemptions are especially detrimental to performance and randomness in CPU scheduling help mitigate intra-program contentions on shared resources. Inspired by these findings, we developed two simple optimizations, *delayed preemption* and *differential scheduling*, at the hypervisor to improve parallel performance under interference. Results show that the two optimizations derived from our analysis with the synthetic workloads are effective in reducing performance slowdowns when real parallel applications are co-located.

3.5.1 Delayed Preemption

Harmful preemptions happen when parallel programs co-locate with periodic or intermittent interference. The frequent wakeups of interfering vCPUs can cause longer steal time and synchronization time in the parallel VM. While preemptions due to the expiration of CPU time quantum are necessary for fair allocation, the boosted wakeups are needed for minimizing latencies of interactive or IO workloads. In a contended environment, such premature preemptions may cause cascading performance degradations in parallel programs. Figure 3.7 suggests that even CPU is heavily contended, there still exists idle time in which both parallel threads and the interfering loops are in blocked (or sleep) state. When these vCPUs wake up, preemptions between the two competing sides cause ping-pong scheduling. To address this issue, we propose *delayed preemption* (DP) to overlap computations with blocking/sleeping, and to minimize premature preemptions.

Inspired by the design of hybrid synchronization, which uses a spin-then-block approach to attain a balance between low latency and wasting CPU time, our approach temporarily delays a wakeup vCPUs for a short period of time in the hope that current running vCPU would voluntarily yield CPU due to waiting for synchronization. Our implementation in Xen is quite simple and the change only consists of 50 lines of code. We added a single shot timer to the pCPU that has a waking vCPU. If the current running vCPU voluntarily yields CPU, the timer is stopped. Otherwise, the expiration of the timer forces a call to the `schedule` function in Xen to preempt the current vCPU. However, the selection of the delayed period is challenging as different applications attain the best performance with different delays.

Figure 3.9 shows the performance of PARSEC benchmarks due to stock Xen and DP. We co-located PARSEC benchmarks with two background workloads. *streamcluster* has fine-grained synchronizations at the granularity of 20-30ms while *fluidanimate* has coarse-grained synchronizations every 6 seconds. Benchmarks in Figure 3.9(a) suffer involuntary preemptions caused by the background *streamcluster* and the background *fluidanimate*

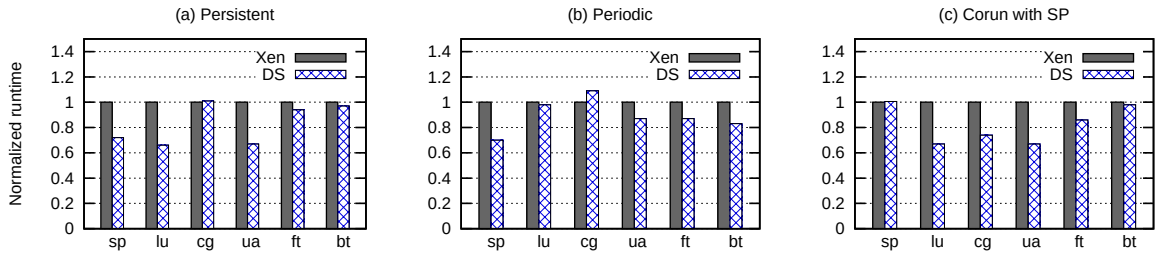


Figure 3.10: Performance of NPB benchmarks under *differential scheduling* (DS).

in Figure 3.9(b) is the victim of premature preemptions. In general, the optimal delay setting varies across benchmarks. As shown in Figure 3.9(a), short delay (i.e., DP(2ms)) is more desirable for *streamcluster*, *facesim*, *dedup*, and *bodytrack*, while long delay (i.e., DP(8ms)) worked best for other workloads. Similarly, workloads also have their respective preferences of delay in Figure 3.9(b).

To meet applications' diverse needs of preemption delay, we make DP adaptive (i.e., *adaptive-DP*). As discussed in Section 3.3, the *idle* time on pCPUs is a key indicator of the efficiency of parallel scheduling. For PARSEC benchmarks, vCPUs become idle/blocked when synchronizing with sibling vCPUs. Interference could slowdown sibling vCPUs and prolong the idle period. In multiprocessor scheduling, uncooperative preemptions could leave CPU time on pCPUs unused when vCPUs from competing workloads are all blocked by synchronization. The objective of adaptive-DP is to dynamically adjust the preemption delays for co-running parallel workloads so that the overall preemptions and the idle time on pCPUs are minimized. We decompose runtime into *compute* and *idle* time and the latter includes *sync* and *steal* time. We adaptively change the preemption delay for a vCPU until its compute time approximates its expected fair share in a shared environment. If all vCPUs from co-running applications attain their fair share, the idle time would be dominated by the steal time, thereby minimizing the sync time and preemptions. Specifically, the preemption delay is updated as follows:

$$lag = \frac{t_{comp} - t_{fair}}{t_{comp} + t_{idle}}, \tag{3.1}$$

$$delay = delay + lag \times (delay + \texttt{MICROSECS(10)}), \tag{3.2}$$

where t_{fair} and t_{comp} refer to the ideal fair allocation of CPU time and the actual attained compute time, respectively. The preemption $delay$ is updated according to the lag relative to the ideal fair CPU allocation. A positive lag increases the delay a vCPU waits to preempt the current running vCPU, giving other applications more time to execute. `MICROSECS(10)` is to ensure no-zero changes in case delay becomes zero.

As shown in Figure 3.9 (a) and (b), *adaptive-DP* automatically determines the optimal delay for different applications. When co-running with fine-grained *streamcluster* (Figure 3.9 (a)), adaptive-DP outperformed both DP (2ms) and DP (8ms) for applications with fine-grained synchronizations, e.g., *streamcluster*, *facesim*, and *canneal*. Adaptive-DP effectively identified better delay values for such workloads compared to the manually determined 2ms and 8ms delays. For coarse-grained workloads, such as *swaptions*, *blackscholes*, and *x264*, adaptive-DP struck a balance between the foreground and background workloads and achieved performance in-between the manually tuned delays. When co-running with *fluidanimate* (Figure 3.9 (b)), adaptive-DP had similar performance to stock Xen as Xen with zero preemption delays always prioritized the foreground workloads with fine-grained synchronization. We calculated the geometric mean of the slowdowns of the foreground and background workloads relative to their performance in dedicated systems. The background workloads were repeated until all foreground workload completed. Adaptive-DP outperformed Xen by 12% in the overall slowdown, indicating more efficient scheduling in a shared environment.

3.5.2 Differential Scheduling

Differential Scheduling (DS) is also motivated by our observations in Section 3.3. We found that the randomness in intermittent interference helps alleviate the contention on shared memory resources, which significantly reduces the required computation. The irregularity in CPU demand and sleep intervals forces the CPU scheduling on multiprocessors to proceed at different paces. This effectively leads to different lengths of time slice at different CPUs because the intermittent loops yield CPU at irregular intervals. The result is not only the release of pressure placed by the concurrent vCPUs to the memory hierarchy but also a reduction of wasted spinning time due to fine-grained scheduling.

To emulate the benefits brought by co-running with intermittent interference, we purposely make the schedulers on multiple CPUs have different time quantum. Xen uses a master timer for each pCPU to generate periodic timer interrupt to force a call to the `schedule` function. The default timeout (i.e., the time slice) is 30ms. In DS, the interval of the timer is randomly generated. Each time Xen sets the timeout for the next timer interrupt, it picks a random interval based on the readings of the Time Stamp Counter (tsc) register. Given the micro-second resolution (approximately 2430 cycles on our platform) of hardware timers, the last two digits of the tsc readings are likely device noises and are a good source of randomness. We set the time quantum on individual pCPUs to the range of 10ms to 30ms to enable fine-grained scheduling.

Figure 3.10 shows the performance of NPB benchmarks due to stock Xen and DS. Since DS already adds randomness into vCPU scheduling, we present the performance of DS when NPB benchmarks ran with persistent, periodic interferences (excluding the intermittent interference), and the real workload *sp*. The selection of *sp* is due to its memory contentiousness to co-running programs. Figure 3.10 (a) suggests that with persistent interference, DS was only effective in optimizing memory-bound programs, such as *sp*, *lu*, and *ua*. On average, DS outperformed Xen by 32% in these workloads. For CPU-bound program, e.g., *cg*, DS neither significantly degraded or improved performance. For periodic interference (shown in Figure 3.10 (b)), DS outperformed Xen in all benchmarks except *cg*. DS does not help mitigate memory contentions as *cg* is primarily CPU-bound and hurts performance because differential time slices slow down the tightly coupled phases in *cg*.

Interestingly, DS was able to outperform Xen even for compute-bound benchmarks, e.g., *cg* (as shown in Figure 3.10 (c)) when the background workload was memory-bound *sp*. The alleviation of memory pressure from the background *sp* helped improve the performance of foreground workloads. However, the inability of DS to improve foreground *sp* performance in Figure 3.10 (c) suggests that DS work best for workloads with complementary memory access patterns. Another advantage of DS is that the randomness in scheduling significantly reduces runtime variations across runs, with an average variation of 0.5%.

3.6 Summary

Enable per-thread runtime breakdown Currently, vProfile can only report runtime breakdown at per-vCPU level for parallel applications that have a one-to-one mapping from user-level threads to vCPUs. There exist many parallel workloads with more threads than vCPUs. Examples include web servers and workloads implementing worker pools, such as *dedup*, *ferret* and *vips*. Enabling per-thread runtime breakdown would help further pinpoint the source of slowdown but require tracing thread switches in the guest OS.

Extend prediction to more sophisticated workloads The online prediction currently focuses on multithreaded parallel programs on shared memory systems. Predicting parallel performance on distributed memory systems present significant challenges. Much effort is needed to devise a lightweight and accurate sampling approach on multiple machines. Further, our prediction treats the useful work on any threads equally. For more sophisticated programs, especially those having strong dependencies between threads, e.g., *dedup* and *ferret* with pipeline parallelism, the prediction should be based on the useful work of the most critical thread.

Identify symbiotic workloads Experimental results show that DP and DS are effective for blocking-based workloads (e.g., PARSEC benchmarks) and workloads with complementary memory demands, respectively. However, no one size fits all. DS can hurt the performance of CPU-bound and tightly bounded workloads, and DP is ineffective for applications with busy-waiting synchronization. Identifying symbiotic workloads that either have complementary resource demands or can be managed under similar schemes would help improve resource utilizations and reduce energy consumptions in datacenters.

This chapter presents a systematic study of parallel performance under interference. We find that the speed of individual threads under interference is determined by their varying resilience to interferences and the computation required to complete the parallel program can change vastly under interference due to alleviated intra-program contentions. Further, the overall performance is the result of the complex interplays between these factors. Avoiding harmful vCPU preemptions or maintaining asynchrony between vCPUs helps reduce slowdown under interference for different kinds of workloads. Inspired by these findings, we develop an accurate online approach for predicting slowdowns under interference without requiring completing the parallel program, and devise two scheduling optimizations at the hypervisor to improve performance.

CHAPTER 4

Interference-Resilient SMP Virtual Machine Scheduling

4.1 Introduction

Symmetric Multiprocessing virtual machines (VMs) are becoming increasingly common in cloud datacenters. To fully utilize hardware parallelism, SMP VMs are often used by cloud users to host multi-threaded applications. On the other hand, cloud providers prefer oversubscribing their datacenters by consolidating multiple independent VMs onto a single machine to improve hardware utilization and reduce energy consumptions. For example, in desktop virtualization, VMware suggests a physical CPU (pCPU) can be shared by as many as 8 to 10 virtual desktops [107]. However, oversubscription requires that the CPU be multiplexed among multiple VMs so that each VM receives only a portion of the pCPU cycles.

CPU oversubscription introduces challenges to efficiently executing parallel and multi-threaded programs in SMP VMs. One well-known issue is the lock-holder preemption (LHP) problem [36]. LHP occurs when a vCPU is descheduled by the hypervisor while the thread currently running on that vCPU is holding an important lock. As the performance of parallel applications as a whole depends critically on the cooperation of multiple threads, if one thread holding the lock is preempted, other threads waiting for the lock are unable to make progress until the descheduled vCPU is rescheduled. Thus, the delay of one vCPU will significantly degrade the overall performance of the parallel program. Lock-waiter

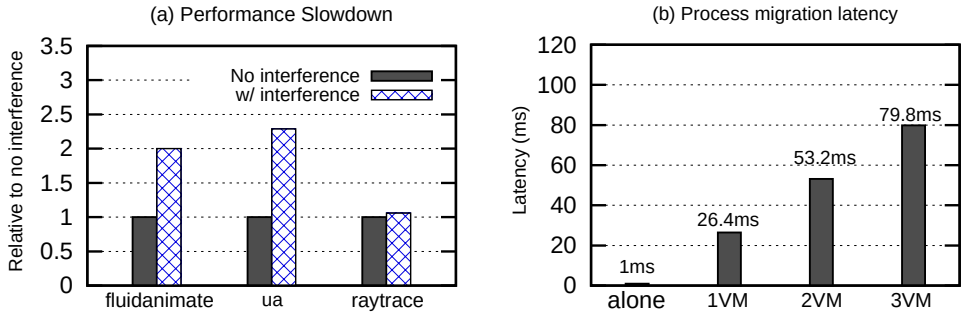


Figure 4.1: LHP and LWP cause significant slowdown to parallel programs. (a) Programs with user-level load balancing are more resilient to interference. (b) Existing load balancing in the guest OS is ineffective in addressing LHP and LWP.

preemption (LWP) [52, 3, 101] is a similar problem in virtualized environments and can cause severe slowdown.

The root cause of the LHP and LWP problems is the semantic gap between the guest OS and the hypervisor. In virtualized environments, there exist two scheduling domains: (1) the guest OS schedules processes on vCPUs and (2) the hypervisor schedules vCPUs on physical CPUs. The scheduling activities in the guest OS are completely oblivious to the hypervisor. Thus, a vCPU can be preempted at any time by the hypervisor regardless of what this vCPU is executing. If vCPUs with threads waiting for entering into or already in the critical sections are preempted, LWP and LHP will occur respectively. There have been studies narrowing the semantic gap by inferring scheduling events inside VMs at the hypervisor using heuristics [49, 55, 73, 85, 86, 110], or approximating VM co-scheduling to mitigate the LHP and LWP problems [59, 102, 23, 99], or allowing the guest OS to assist hypervisor scheduling [105, 36, 30, 115, 21, 78]. These approaches have their respective limitations. Different workloads require distinct heuristics to identify thread criticality; co-scheduling is expensive to implement and causes CPU fragmentation [93]; synchronization-oriented optimizations make the hypervisor scheduling very complex and can possibly compromise fairness between VMs.

In this work, we identify another semantic gap, which is neglected in the literature – the guest OS is also unaware of the scheduling events at the hypervisor. If this gap is bridged, the guest OS can proactively migrate a critical thread if the host vCPU is preempted. We ran parallel programs in a 4-vCPU VM and slowed down one vCPU by co-locating another compute-bound VM with the vCPU to create interference. The interfered vCPU had frequent LHPs and LWPs. Figure 5.1 (a) shows the performance slowdown of three parallel application. `Fluidanimate` from the PARSEC benchmarks [103] and `ua` from the NPB benchmark [8] use blocking and spinning synchronization, respectively, and had significant slowdowns. In contrast, `raytrace` was resilient to LHP and LWP due to its user-level load balancing, which absorbed the slowdown by distributing work to threads having no interference.

Modern OSes are equipped with complex load balancing schemes to efficiently utilize multiprocessor systems. However, load balancing in the guest OS is not effective in virtualized environments. Process migration is the critical operation in load balancing. Figure 5.1(b) shows the latency of migrating a process in a Xen VM from a vCPU with frequent preemptions to another vCPU without interference. We measured the average latency of 30 migrations. The frequency of preemption was managed by placing different numbers of compute-bound VMs with the source vCPU of the migration. The reference migration latency was obtained when the VM ran alone and there were no vCPU preemptions. Figure 5.1 (b) suggests that process migration latency increases with the level of contention

on the vCPU and the latency jump at each step corresponds to the VM scheduling delay (i.e., 30ms in Xen credit scheduler [10, 118]) incurred by adding one more VM. The results infer that load balancing in the guest OS is unable to address the LHP and LWP problems and itself is affected by vCPU preemptions.

There are two reasons why in-guest load balancing does not help mitigate LHP and LWP, both of which are due to the unawareness of hypervisor scheduling events in the guest OS. First, vCPU preemptions do not cause load imbalance in the guest, thereby the guest is unable to invoke process migration. Second, threads on preempted vCPUs are in "running" state, though they are actually not running, to the guest OS. As a result, the guest OS fails to migrate such "running" threads because it thinks it is unnecessary. Process migrations will only be successful until the preempted vCPU is scheduled again.

To fully unlock the potential of the guest OS in addressing the LHP and LWP problems, we design *interference-resilient scheduling* (IRS), a simple approach to bridging the guest-hypervisor semantic gap and guiding guest load balancing. Inspired by scheduler activations (SA) [5] in hybrid threading, IRS notifies the guest OS and activates in-guest load balancing when a vCPU is to be preempted by the hypervisor. As such, lock holder threads can be promptly migrated to other running vCPUs to avoid LHP and LWP.

We have implemented a prototype of IRS in Xen 4.5.0 and Linux 3.18.4, and performed comprehensive evaluations with various parallel and multi-threaded workloads. Experimental results show that IRS can improve the performance of NPB and PARSEC benchmarks by up to 43% and 42%, respectively, especially for programs with heavy synchronization. Moreover, IRS can reduce the latency of multi-threaded server workloads by as much as 46%.

The rest of the chapter is organized as follows. Section 4.2 discusses previous work on the LHP and LWP problems and presents our motivation. Section 4.3 and 4.4 describe the design and implementation of IRS, respectively. Evaluation results and analysis with various parallel applications are given in Section 5.4. Section 4.6 discusses limitations and future work. We conclude this chapter in Section 4.6.

4.2 Motivation

As discussed above, existing work, either the hypervisor-level or guest OS-assisted approach, focused on making the hypervisor aware of the synchronization event inside the guest OS to aid scheduling. We show that there is a great potential of the guest OS to address the LHP and LWP problems.

Potential of guest OS load balancing We ran representative parallel benchmarks from the PARSEC and NBP benchmark suites in a 4-vCPU Xen VM. The LHP and LWP problems

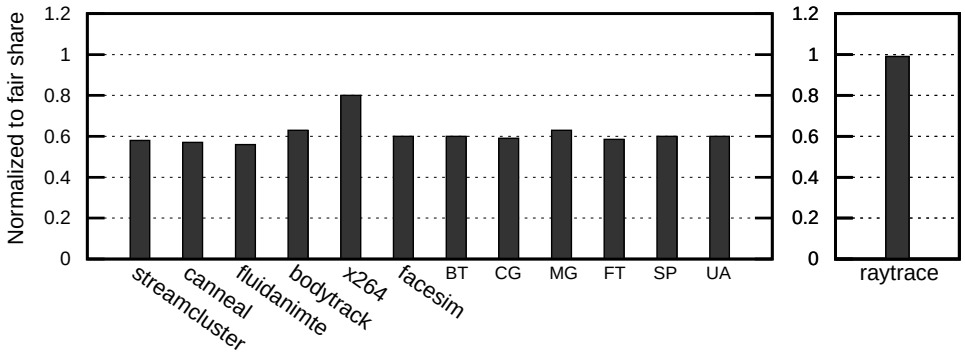


Figure 4.2: Parallel applications suffer low CPU utilization due to interference. User-level load balancing helps efficiently utilize CPU under interference.

were caused by placing another CPU-bound VM with one of the 4 vCPUs. The vCPUs of the parallel VM and the interfering VM were pinned to separated pCPUs. The vCPU that experienced LHP and LWP shared the same pCPU with the interfering VM. The PARSEC benchmarks were compiled with `pthreads` and NBP benchmarks were compiled using `OpenMP` with `OMP_WAIT_POLICY` set to passive. All benchmarks used blocking synchronization.

Figure 4.2 shows the CPU utilization of the parallel VM relative to its fair share. Ideally, both VMs should receive the fair share of the pCPU capacity. As shown in the figure, all parallel programs except `raytrace` suffered much lower CPU utilizations compared to their fair shares, indicating that the parallel VM did not fully or efficiently utilize its CPU entitlement. The culprit is that the interfering VM caused frequent LHPs and LWPs to the parallel VM. If a critical thread is preempted, all other threads need to wait for the critical section until the preempted vCPU is rescheduled. With blocking synchronization, the waiting threads are put to sleep and their host vCPUs become idle even when there are sufficient pCPU allocated to the parallel VM.

Programs with spinning synchronization suffer similar performance degradation due to LHP and LWP, but do not show low CPU utilizations. Instead of going idle, the waiting vCPUs busily wait on the lock and burn CPU cycles. Although the parallel VM is able to utilize its fair share, most CPU cycles are spent on spinning and few are used to carry out meaningful computation. Hardware-based techniques, such as pause loop exiting (PLE) [44], detect excessive spinning and stop a VM to prevent it from wasting CPU cycles. The effect is equivalent to blocking-based synchronization and the parallel VM will suffer low CPU utilization in the presence of LHP and LWP.

In contrast, Figure 4.2 also shows that `raytrace` was able to fully use its fair share even in the presence of LHP and LWP. This explains its resilience to interference

as shown in Figure 5.1. `Raytrace` implements a work-stealing mechanism at user level and threads that complete their assigned work sooner steal the work originally assigned to slower threads. As such, interference has less impact on the overall performance as meaningful work is migrated to faster or interference-free threads/vCPUs. Similar interference resilience can also be observed in programs compiled with Intel TBB [45] and `OpenMP` using a dynamic thread schedule.

This motivating example demonstrates that load balancing can effectively address the LHP and LWP problems and mitigate the slowdown caused by interference. However, only programs that have specific compiler support or have their own user-level load balancing are resilient to interference. Programs relying on the guest OS, e.g., Linux, for load balancing suffer low CPU utilizations and significant performance slowdowns. In general, there are two approaches in guest OS load balancing: *push migration* and *pull migration*. Push migration periodically checks **load imbalance** and pushes threads from busy to less-busy vCPUs; pull migration occurs when a vCPU becomes idle and steals (or pulls) **excessive** work or ready (but not running) threads from a busy vCPU. Both approaches fail to work effectively in virtualized environments. First, the load imbalance at the hypervisor does not lead to imbalance in the guest OS and push migration is not invoked. Second and most importantly, threads on preempted vCPUs are not considered excessive work by the pull migration as they are in the "running" state.

Issues with hypervisor load balancing Hypervisors also implement complex schemes for balancing vCPUs among pCPUs. Hypervisor level load balancing falls short of addressing the LHP and LWP problems in two ways. First, lacking the information on thread criticality, the hypervisor is unable to precisely identify the vCPU that experiences LHP and LWP. Second, the hypervisor treats vCPUs from different VMs equally and relies purely on the computational load on pCPUs for load balancing. Thus, it is possible that hypervisor places vCPUs from the same VM onto the same pCPU to attain better load balance, thereby causing the CPU stacking problem [99]. Our experimental results show that CPU stacking can incur 10-20x performance degradation to PARSEC benchmarks when the parallel VM and the interfering VM shared the same set of 4 pCPUs but all vCPUs were unpinned. The same issue can also be observed in other hypervisors, such as KVM [54] and VMware [102].

Summary Parallel programs suffer significant performance loss due to LHP and LWP and so are unable to efficiently utilize their CPU allocations. Effective load balancing of parallel threads can greatly alleviate the LHP and LWP problems. These observations motivated us to enhance the guest OS load balancing in virtualized environments so as to make any workload resilient to interference. To this end, we design *interference-resilient scheduling* (IRS), a simple approach to bridging the guest-hypervisor semantic gap and unlocking guest OS load balancing.

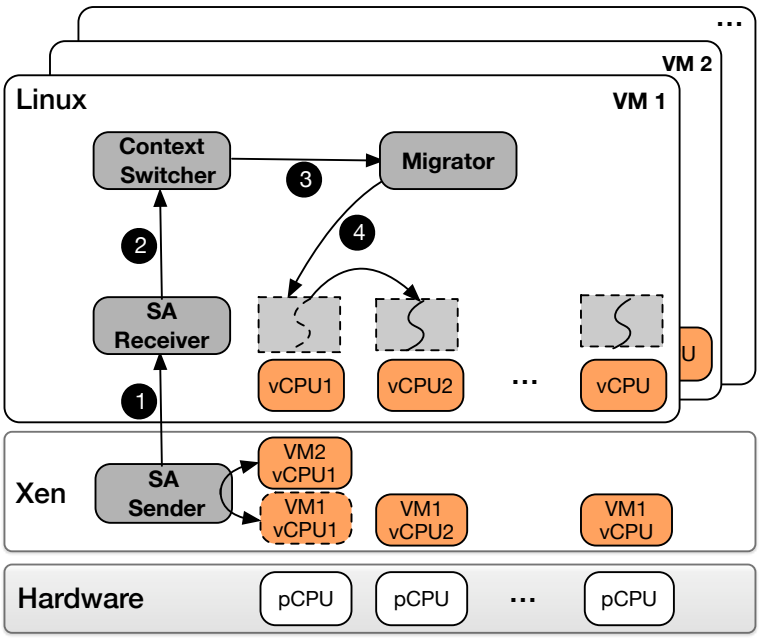


Figure 4.3: The architecture of IRS.

4.3 IRS Design

IRS is a coordinated approach that bridges the guest-hypervisor semantic gap at the guest OS side. The objective is to enhance the guest OS load balancing to make parallel programs resilient to interference between VMs, thereby mitigating the LHP and LWP problems. The heart of IRS design is the mechanism of *scheduler activations* (SA) in response to vCPU preemptions at the hypervisor. Inspired by the classical scheduler activation approach in hybrid threading, in which the OS kernel notifies the user-level scheduler if a user-level thread blocks in the kernel so that the user-level scheduler can pick another ready user thread to execute. Similarly, IRS informs the guest OS once its vCPU is to be preempted. The guest OS then migrates the thread running on the preempted vCPU to another running vCPU to avoid LHP and LWP.

Figure 4.3 shows the architecture of IRS in a Xen environment. There are four components in IRS: *SA sender*, *SA receiver*, *context switcher* (CS), and *migrator*. Before Xen preempts a vCPU, it sends a notification to the vCPU via SA sender residing in Xen (step ①). Upon receiving the notification, SA receiver in the guest starts the load balancing process (step ②). To enable task migration, the CS deschedules the thread on the preemptee vCPU and marks the thread as migrating (step ③). Last, the migrator moves the thread to a sibling vCPU with least waiting time (step ④). Next, we elaborate on the design of these components in the context of Xen and Linux guest OS.

Algorithm 1 Send and acknowledge SA event.

```
1:  Variables: The vCPU to be preempted v; The SA acknowledgement sent by the guest
    OS ops.
2:  /* Hypervisor: send SA event */
3:  procedure Send_SA_event(v)
4:      if vcpu_runnable(v) and sa_pending(v) then
5:          send_guest_vcpu_virq(v_c, VIRQ_SA_UPCALL)
6:          set_sa_pending(v)
7:          return continue_running(v)
8:  /* Guest OS: acknowledge SA completion */
9:  procedure Ack_SA_event(void)
10:     ops = context_switcher()
11:     wake_up_migrator()
12:     /* Return the control back to hypervisor and clear SA pending flag on the host
    vCPU */
13:     HYPERVISOR_sched_op(ops, NULL)
```

4.3.1 SA Sender and Receiver

SA sender and receiver together establish a communication channel between the hypervisor and the guest OS. Algorithm 1 shows the interactions between the SA sender and receiver. SA sender is on the critical schedule path of the hypervisor. Whenever the hypervisor decides to preempt a current running vCPU, it sends a notification to the preemptee vCPU to allow the guest OS to respond to the preemption. Only vCPUs that are involuntarily preempted and are still willing to run (i.e., `runnable`) will be notified (line 4-5). To avoid duplicate notification, the SA sender also needs to check if there is an SA notification pending for a vCPU in the guest OS. The SA notification is per-vCPU. After the notification is sent, the hypervisor delays the preemption and allows the preemptee vCPU to continue running and process the notification (line 7).

The SA receiver resides in the guest OS and takes three steps to respond to the SA notification: (1) deschedule the current running task on the preemptee vCPU and perform a context switch (line 12). The return value of the context switcher determines the response to the hypervisor; (2) asynchronously wake up the migrator thread to move the descheduled task to a different vCPU (line 13); (3) return the control back to the hypervisor (line 15). Once the hypervisor receives the response, it clears the SA pending flag of the vCPU to enable the next round of SA.

The hypervisor-guest communication uses Xen's event channel for SA notification. To ensure timely delivery of the SA, we design the notification as a virtual interrupt (vIRQ) for the guest OS. The SA receiver is essentially the interrupt handler of the new vIRQ. Note that one change to hypervisor level scheduling is necessary for enabling SA – any vCPU preemption needs to be delayed until the guest OS completes the processing of SA. This change may affect existing scheduling in the hypervisor, such as fairness and I/O prioritization. To minimize the impact, the SA receiver should complete fast. The context switching of the current running task should be performed on the preemptee vCPU and the vCPU needs to be active. Once the context switch is done, the migrator is asynchronously invoked and can run on other vCPUs. Thus, the required delay at the hypervisor only includes the time to handle the vIRQ and perform one task context switch in the guest. Our profiling suggests that IRS adds 20-26 μs delay to the hypervisor scheduling. Since the time slice of hypervisor scheduling is in the granularity of milliseconds, e.g., 30ms in Xen, 6ms in KVM, and 50ms in VMware, the delay is negligible from the perspective of fair CPU allocation. However, if vCPU preemption is due to prioritizing an I/O-bound vCPU, the delay will add to I/O latency.

4.3.2 Context Switcher

The purpose of the context switcher is to faithfully reflect the status of a vCPU in the guest OS to bridge the semantic gap. For example, if a vCPU is preempted and put back to the runqueue of a pCPU, the task currently running on the vCPU in the guest OS should also be descheduled. After a context switch, the vCPU should be put into a proper state so as not to affect hypervisor-level scheduling. In Xen, vCPUs are in one of the following three states: `running`, `runnable`, and `blocked`. While `running` means a vCPU is executing on the pCPU, `runnable` indicates that the vCPU has been preempted but it has a task to run. If a vCPU is idle or waiting for I/O completion, it has no tasks to run and will be put in the `blocked` state. Xen devises different scheduling policies for different vCPU states. For example, a vCPU waking up from a `blocked` state will be considered latency sensitive and be prioritized.

To preserve the scheduling policy at the hypervisor, the SA receiver should respond differently to the hypervisor depending on the execution state of the vCPU after task context switch. The context switcher returns different operations to the SA receiver (Algorithm 1, line 12). If there is no runnable task left in the runqueue of the vCPU after the current running task is descheduled, the `idle` task will be put on the vCPU and the context switcher returns an operation `SCHEDOP_block`. In contrast, if there are other runnable tasks in the runqueue of vCPU, it should be put in the `runnable` state in hypervisor. In

Algorithm 2 Migrate task from preempted vCPU.

```
1:  Variables: The task to be migrated p; the least loaded vCPU v_min in the guest OS; the
    state of a vCPU s; the runqueue of a vCPU rq_v.
2:  /* Guest OS: migrate task to least loaded vCPU */
3:  procedure Migrate_task(p)
4:      v_min = NULL
5:      for each online vCPU v do
6:          rq_min = rq_of(v_min)
7:          s = get_vcpu_runstate(v)
8:          if s == IDLE then
9:              v_min = v
10:             break
11:         if s == RUNNING then
12:             rq_v = rq_of(v)
13:             if rq_v.rt_avg < rq_min.rt_avg then
14:                 v_min = v
15:     if v_min ≠ NULL then
16:         __migrate_task(p, v_min)
17:         return SUCCESS
18:     else
19:         return FAIL
```

this case, context switcher returns an operation `SCHEDOP_yield`, which does not change the vCPU state but simply yields to the hypervisor.

4.3.3 Migrator

The migrator is responsible for distributing the descheduled task from a preempted vCPU to another running vCPU so that the task does not need to wait for the original vCPU to be scheduled so as to run. If the descheduled task is a lock holder or lock waiter and is scheduled sooner due to load balancing, the LHP and LWP problems are alleviated. As discussed in Section 4.2, load balancing in the guest OS is not effective due to the two semantic gaps: load imbalance at hypervisor does not trigger load balancing in guest OS; task migration does not apply to "running" tasks even though the host vCPU is preempted. The context switcher addresses the second gap by descheduling the task upon vCPU preemption. The migrator bridges the first gap.

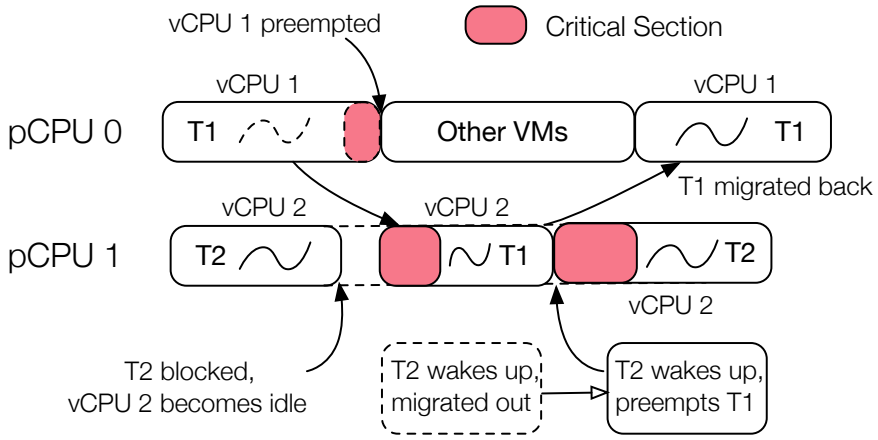


Figure 4.4: Pingpong migration caused by IRS and a simple approach to preserve locality.

Since the guest OS, e.g., Linux, has implemented complex load balancing schemes, we design the migrator to be minimally intrusive to the existing balancing algorithm. Since the guest load balancer is unable to sense the load imbalance at the hypervisor, the migrator does not consider the load balance in the guest and forcibly move the descheduled task to a different vCPU. The goal is to migrate the task to the least loaded vCPU. Algorithm 2 shows how to find the least loaded vCPU. The migrator iterates over all online vCPUs of the guest OS until it finds a target vCPU for migration. Note that preempted vCPUs also appear to be "online" to the guest OS. Therefore, the migrator needs to call down to the hypervisor to check the actual vCPU state (line 7). Ideally, the migrator finds an idle sibling vCPU and the search will end as the task can run immediately on this vCPU (line 8-10).

If there are no idle vCPUs, the migrator tries to find the least loaded vCPU (line 12-17). As there are two levels of load balance in virtualized systems, i.e., the balance in the guest OS and in the hypervisor, the migrator aims to find a lightly loaded vCPU, which not only has few tasks on the vCPU runqueue in the guest but also experiences little contention from other VMs on the pCPU. We rely on the real time estimate of runqueue load (i.e., `rt_avg`) in Linux to measure vCPU busyness. The metric `rt_avg` considers the weighted process load of each vCPU in the guest OS as well as the contention experienced by this vCPU on the pCPUs. It uses `steal` time, which measures the time during which a vCPU is runnable but unable to run on the pcpu due to contention, to quantify hypervisor-level CPU contention. The migrator compares vCPUs using the `rt_avg` of their runqueues to pick the least loaded vCPU.

Another challenge in designing the migrator is to ensure that load is balanced between sibling vCPUs when preempted vCPUs come back online. To minimize intrusive changes to the guest OS, the migrator relies on the existing load balancer in Linux to move

tasks back to the rescheduled vCPU. However, one drawback of task migration is the loss of cache locality. The migrator aims to preserve cache locality as much as possible. Besides the push and pull migrations in Linux, there is another scenario in which task migration is necessary and related to parallel programs. For workloads with blocking synchronization, such as `pthread mutex` and `barrier`, the Linux kernel checks load balance when waking up waiting threads, e.g., lock waiters. If the vCPU where the waiting thread slept on is running another task, the waking task is migrated to a different vCPU. Figure 4.4 illustrates this problem and the migrator's simple solution.

As shown in Figure 4.4, when vCPU-1 is preempted, task-1 (T1) is migrated to idle vCPU-2, on which task-2 (T2) is blocked and waiting for the lock held by T1. Once T1 releases the lock, T2 is woken up. Because T1 is currently running on vCPU-2, T2's the host vCPU, T2 will be migrated out, likely to vCPU-1 as it is idle now. This design is to avoid unnecessary preemptions of a running task if there exist idle vCPUs. However, the wake up balancing causes pingpong migrations between vCPUs, which leads to poor cache locality. Waking tasks are frequently migrated away from their original vCPU because the migrator distributes tasks from preempted vCPUs to idle vCPUs.

The migrator employs a simple approach to address this issue. Instead of migrating the waking task, the wakeup load balancer in Linux is modified to check the status of the current running task to determine if the waking task should preempt the current task. The migrator tags each task that is migrated due to preempted vCPU. If the current running task is tagged, the wakeup balancer allows the waking task to preempt the current task. The dotted box in Figure 4.4 shows the original Linux design and the arrow points to the new design. This simple solution guarantees that waiter tasks always wake up from their host vCPU to preserve locality. We rely on the Linux load balancer to migrate the tagged task back to the preempted vCPU when it is scheduled again. This design only applies to blocking workloads. For spinning workloads, the Linux balancer will migrate the tagged task back to its original vCPU as its runtime on the new vCPU is short and it is not "cache hot".

4.4 Implementation

We have implemented IRS in Xen 4.5.0 and Linux 3.18.4. We intend to make the changes to the hypervisor and guest OS minimally intrusive and use existing scheduling and load balancing primitives. IRS requires small changes to Xen (less than 30 lines of code) and Linux guest kernel (about 130 lines of code). Next, we describe the modifications to Xen and Linux in detail.

4.4.1 Modifications to Xen Hypervisor

For SA notification, we add a new virtual interrupt `VIRQ_SA_UPCALL` in Xen and use a dedicated event channel for SA communications between Xen and the guest OS. The credit scheduler in Xen is modified to temporarily delay the preemption of vCPUs until the guest OS acknowledges the completion of SA. Once Xen relinquishes the control of the vCPU scheduling, it relies on the guest OS to respond to the SA notification and return the control back to Xen. This may create security issues if malicious guests never return to the hypervisor. As discussed in Section 4.3.1, SA processing typically takes 20-26 μs, so the hypervisor can set a hard limit for SA completion to prevent rogue users from exploiting SA.

4.4.2 Modifications to Linux Guest OS

The main functionalities of IRS are implemented in the guest OS. We implement SA receiver as the interrupt handler of the new `VIRQ_SA_UPCALL` interrupt. Since interrupt handlers should be kept small, SA receiver delegates the SA response to Xen to the context switcher. We implement the context switcher as the bottom half of the `VIRQ_SA_UPCALL` vIRQ. We create a new softirq called `UPCALL_SOFTIRQ` in the guest OS and assigned the context switcher as its handler. In Linux, softirqs have different priorities. We set the `UPCALL_SOFTRIQ` to a lower priority than the `TIMER_SOFTIRQ`, which is responsible for handling periodic timer events because the Linux kernel relies critically on timer interrupts to perform task scheduling. When timer interrupt and SA interrupt arrive at the same time, we ensure that the timer interrupt, which may trigger task switching in the Linux scheduler, is handled prior to the SA interrupt. This is to prevent tasks that were to be descheduled at the timer interrupt from being migrated.

The context switcher uses existing scheduling primitives in Linux to pick the next task when the current running task is descheduled. After the context switch is completed, it asynchronously invokes the migrator to distribute the descheduled task to another vCPU for load balancing. Before the migration is performed, the context switcher calls hypercall `HYPERVISOR_sched_op` with either `SCHEDOP_block` or `SCHEDOP_yield` as the command to return control to Xen. The migrator is implemented as a system-wide kernel thread. It borrows the idea from existing migration function `migration_cpu_stop` but need not require to run on the vCPU from where the task is migrated. This greatly shortens the amount of time the preemptee vCPU needs to be active, thereby reducing the delay at the hypervisor scheduler. The migrator probes the runtime states of vCPUs via the hypercall `HYPERVISOR_vcpu_op` to determine the least loaded vCPU for migration. If

a target vCPU is found, the migrator invokes function `__migrate_task` to migrate the task.

4.5 Evaluation

In this section, we present an evaluation of IRS using various parallel and multi-threaded workloads. We study the effectiveness of IRS in improving the performance of various parallel workloads with different types of synchronization (§ 4.5.2). We then extend the evaluation to multi-threaded workloads with little synchronization (§ 4.5.3). We also investigate how well IRS improves overall system efficiency when consolidating multiple parallel workloads (§ 4.5.4) and perform a scalability and sensitivity analysis of IRS in response to various levels of interference (§ 4.5.5). Finally, we study the potential of IRS in mitigating the vCPU stacking problem (§ 4.5.6).

4.5.1 Experimental Settings

Our experiments were performed on a DELL PowerEdge T420 server, equipped with two six-core Intel Xeon E5-2410 1.9GHz processors, 32GB memory, one Gigabit Network card, and a 1TB 7200RPM SATA hard disk. We ran Linux kernel 3.18.4 as the guest and dom0 OS, and Xen 4.5.0 as the hypervisor. We created two VMs, each configured with 4 vCPUs and 4GB memory. One VM was used to run parallel and multi-threaded workloads and the other was the interfering VM. We enabled para-virtualized spin-locks in the guest kernel but it had no effect on NPB performance as OpenMP uses its user-level spin implementation.

CPU pinning We first created a controlled environment to study the benefit of IRS by disabling vCPU load balancing at the hypervisor. Both VMs were set to share four cores in one of the two processors. Each vCPU is pinned to a different pCPU. Thus, two vCPUs from the two VMs share the same pCPU. Note that if vCPUs were unpinned, VM oblivious load balancing at the hypervisor causes CPU stacking problem and incurs significant performance degradation and unpredictability to parallel workloads. In Section 4.5.6, we evaluated IRS performance in an unrestricted environment with all vCPUs unpinned.

Workloads We selected the following workloads and measured their performance with IRS and three representative scheduling strategies.

- *PARSEC* [103] is a shared memory parallel benchmark suite with various blocking synchronization primitives such as mutexes, condition variables and barriers. We compiled them using `pthread` and used the native input.

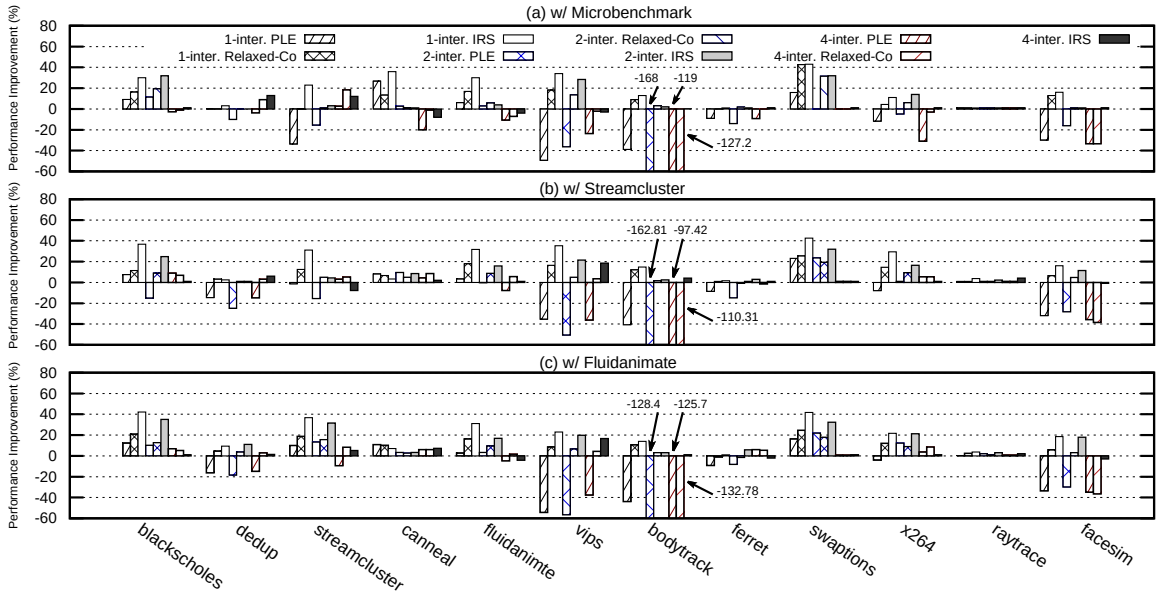


Figure 4.5: Improvement on PARSEC performance (blocking).

- *NASA parallel benchmarks (NPB)* [8] include 9 parallel programs. We used the openMP implementation of benchmarks and set the problem size to class C. Environment variable `OMP_WAIT_POLICY` was set to active to enable spinning synchronization between threads.
- *SPECjbb2005* [94] is a multi-threaded client/server benchmark. Performance is measured by the throughput of the server and the latency of the common request type.
- *Apache HTTP server benchmark* [6] stress tests the throughput and latency of a webserver using a large number of requests. Threads servicing client requests are independent and do not require synchronizations.

Interfering workloads We used two types of interfering workloads to create contention between VMs. We first used a micro-benchmark to generate synthetic interference. The micro-benchmark consisted of a varying number of CPU hogs that compete for the CPU cycles and had almost zero memory footprint. The use of the micro-benchmark is to perform controlled experiments that has persistent interference to the workloads under test. In addition to the micro-benchmark, we also co-located PARSEC and NPB benchmarks with two realistic background interfering workloads respectively. `streamcluster` and `ua` have fine-grained synchronizations at the granularity of 20-30ms and 1-2s while `fluidanimate` and `lu` have coarse-grained synchronizations every 6 and 30 seconds.

Scheduling strategies We compare the performance of IRB with three state-of-the-art scheduling strategies for parallel programs.

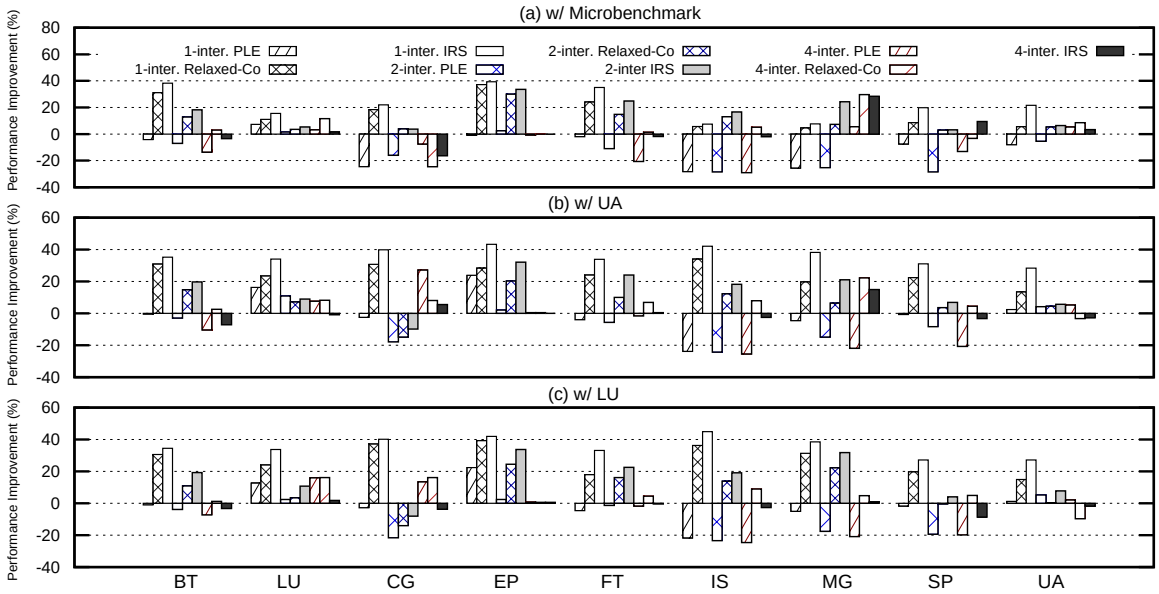


Figure 4.6: Improvement on NPB performance (spinning).

- `Xen`: we used the default credit scheduler without any optimizations for parallel programs as the baseline.
- `PLE`: pause-loop exiting is a hardware-level mechanism for spin detection. It detects the execution of excessive `PAUSE` instructions, which are commonly found in spin lock implementation, and causes trapping (via `VM-exit`) into the hypervisor. In Xen, the credit scheduler switches to a different vCPU if the current vCPU is stopped by PLE. To enable PLE, all workloads were run in hardware-assisted virtualization (HVM) VMs.
- `Relaxed-Co`: we implemented VMWare's relaxed co-scheduling in Xen. `Relaxed-Co` monitors the execution skew of each vCPU and stops the vCPU that makes significantly more progress than the slowest vCPU. A vCPU is considered to make progress when it executes guest instructions or it is in the `IDLE` state. Since VMWare documentation does not reveal further details about relaxed co-scheduling, we implemented an optimization for parallel programs – when a VM's leading vCPU is stopped, the hypervisor switches it with its slowest sibling vCPU to boost the execution of this lagging vCPU.

4.5.2 Improving Parallel Performance

In this section, we evaluate the effectiveness of IRS in improving parallel performance for various parallel workloads. All benchmarks were run with 4 threads, matching the number of vCPUs in the VM. The results were the average of 5 runs.

Figure 4.5 and 4.6 show the performance improvement due to IRS for PARSEC and NPB. Performance improvement is relative to the vanilla Xen and Linux. We varied the level of interference (denoted as *1-inter.*, *2-inter.*, and *4-inter.*) and caused LHP and LWP problems on different numbers of vCPUs of the parallel VM. For example, *2-inter.* refers to the scenario in which either two CPU hogs or 2-thread real applications compete for CPU cycles with two vCPUs of the parallel VM on two pCPUs.

Figure 4.5 shows the effectiveness of IRS for all PARSEC benchmarks. We have the following **observations** about IRS performance:

First, most PARSEC benchmarks benefited from IRS with as much as 42% improvement over vanilla Xen/Linux. However, IRS was not quite effective for some workloads with marginal improvement, i.e., `dedup`, `ferret`, and `raytrace`. `Dedup` and `ferret` employ pipeline parallelism and use multiple threads (i.e., 4 threads) for each pipeline stage (4 stages in `dedup` and 5 stages in `ferret`). Thus, there were multiple threads running on each vCPU. The Linux scheduler was able to balance these threads as most threads will be in the ready state, leaving little room for performance improvement. Similarly, `raytrace` implements user-level load balancing and does not need much help from IRS.

Second, performance improvement decreased as the level of interference increased. While IRS had significantly improved performance for the *1-inter.* and *2-inter.* cases, it can degrade performance in the *4-inter.* case. When a few vCPUs were under interference, IRS was able to migrate threads onto vCPUs without interference. The more interference-free vCPUs, the more likely for IRS to find idle vCPUs that can run migrated threads immediately. In contrast, when all vCPUs were under interference, the vCPU onto which a thread was migrated can be preempted soon, which triggers another round of migration. Frequent migration violates cache locality and may incur performance degradation, especially for memory-intensive workloads. This **overhead** explains the slowdown of some programs under IRS in the *4-inter.* case.

Third, IRS was also effective when interferences were real parallel workloads. The results were similar to those with the synthetic interference except that IRS had slightly better performance in the *4-inter.* case. When the interference was a real parallel program, it demanded less CPU than the synthetic interference because the interfering workload also suffered from LHP or LWP, thereby having low CPU utilizations.

Compared to IRS, `PLE` and `Relaxed-Co` had improved parallel performance to a certain extent, though not as much as IRS in most cases, but incurred considerable performance degradation to some workloads. Since blocking primitives, such as mutex and condition variable, only spend a very short period of time spinning when performing wait queue operations, `PLE` does not help much on preventing excessive spinning. As shown in Figure 4.5, `PLE` had limited performance improvement for `blackscholes` and

swaptions but incurred considerable slowdown to vips, bodytrack, and facesim. The reason is that PLE avoids futile spinning but does not prevent LHP or LWP from occurring. When a spinning vCPU is stopped by PLE, the vCPU from the competing VM will be scheduled. Currently, there is no mechanism in Xen for prioritizing the siblings, which are likely the lock holder or waiter, if a spinning vCPU yields CPU due to PLE. This explains why for some workloads, PLE caused slowdown.

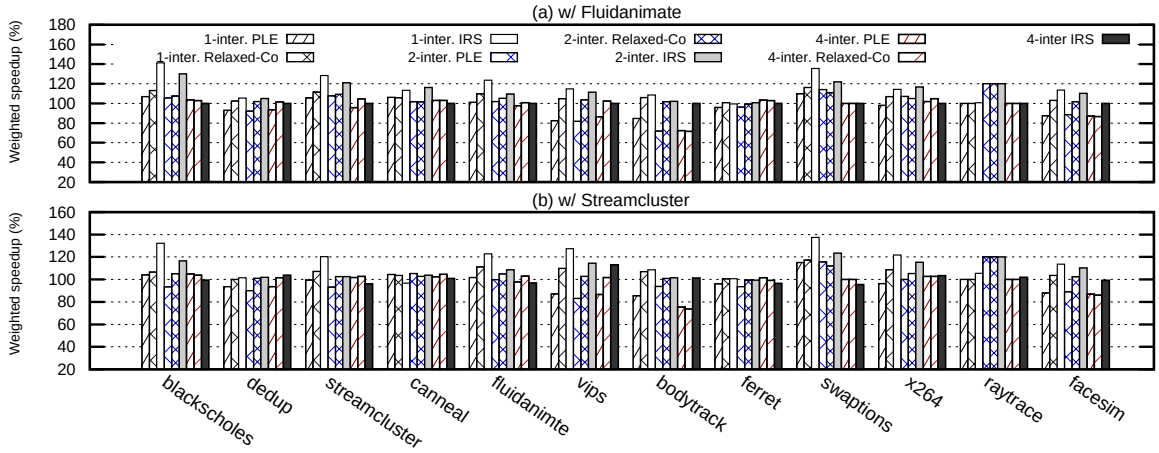


Figure 4.7: Weighted speedup of two PARSEC applications (blocking, higher is better).

In contrast, relaxed-Co is specially designed to balance the progress of sibling vCPUs. In our implementation, we monitored the progress of all sibling vCPUs belonging to the same VM in every accounting period in Xen (every 30ms) and stopped the leading vCPU to boost the most lagging vCPU. However, results in Figure 4.5 show that it attained less performance improvement compared to IRS in almost all PARSEC workloads. The results also suggest that relaxed-Co can be destructive, especially in the *4-inter.* cases.

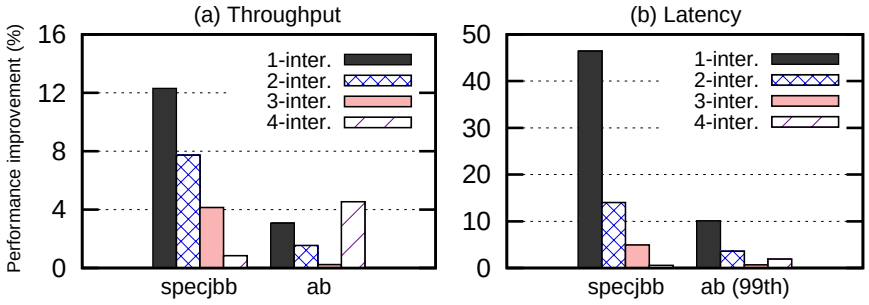


Figure 4.8: Improvement on server throughput and latency.

For example, it caused more than 132% performance degradation for `bodytrack` in Figure 4.5 (c). Overall, `relaxed-Co` was less effective or even destructive for blocking workloads than spinning workloads (shown in Figure 4.6). The culprit was that the idle period during which a blocking workload waits for synchronization is considered as making progress by `relaxed-Co`, thereby not counted as skew. As will be discussed in § 4.5.6, the idleness caused by out of synchronization is not recognized by existing CPU schedulers, which constitutes the main limitation of `relaxed-Co`.

For spinning workloads, the migrator in IRS was unable to find any idle vCPUs to migrate preempted threads as threads never block when waiting for synchronization. IRS can only find vCPUs that are less loaded for migration. Thus, the migrated thread inevitably needs to share the destination vCPU with another thread. Counter-intuitively, Figure 4.6 shows that on average IRS attained higher performance improvement over the baseline. In vanilla Xen/Linux, a preempted thread needs to wait one time slice in Xen, i.e., 30ms, before its host vCPU is scheduled again, leading to long lock wait time. In contrast, IRS migrates the lock holder thread to another vCPU. Although the thread still needs to wait until it is scheduled by the guest OS, the scheduling happens much sooner. Not only does Linux guest OS use finer grained time slices (i.e., 6ms), but also the migrated task likely has smaller virtual runtime than the existing task on the destination vCPU and would be prioritized by Linux completely fair scheduler (CFS). However, a similar trend was observed – the performance gain due to IRS diminished as interference ramped up.

`PLE` and `relaxed-Co` were more effective for spinning workloads than blocking workloads. In most cases, they achieved close but less improvement compared to IRS. Nevertheless, they still performed poorly for some workloads, e.g., `CG`, `IS`, `MG`, and `SP`. In contrast, although IRS can cause slowdowns, the degree of degradation is not as much as `PLE` and `relaxed-Co`.

The improvement on parallel performance was mainly due to much improved CPU utilization under interference. IRS was able to boost the utilization of parallel workloads close to their fair share under CPU contention. The enhanced load balancing in the guest OS helped parallel workloads utilize the idle or wasted CPU cycles in vanilla Xen/Linux.

4.5.3 Improving Multi-threaded Performance

We have shown that IRS is effective for boosting various parallel workloads. In this subsection, we study its performance with more general multi-threaded programs with little or no synchronization. We show that these workloads can also benefit from IRS. We used two different server benchmarks. For *SPECjbb2005*, we set the number of warehouses to 4 so that there was a one-to-one mapping between threads and vCPUs. For *Apache*, we set

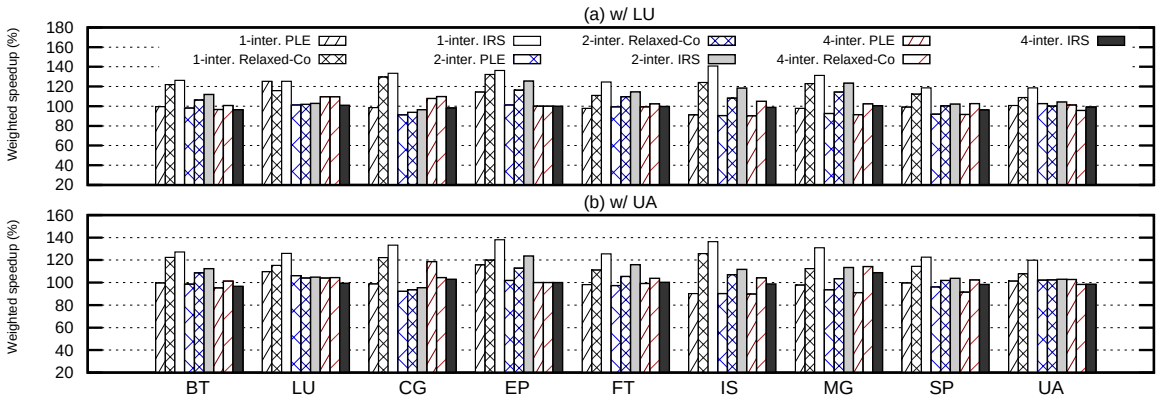


Figure 4.9: Weighted speedup of NPB applications (spinning, higher is better).

the number of connections to 1000 and `MaxClient` in Apache `httpd` to 512. Thus, there were 512 concurrent threads in the webserver.

Figure 4.8 shows the improvement in throughput and latency due to IRS relative to vanilla Xen/Linux. Since request processing in server workloads has little dependency and requires little synchronization, `PLE` and `relaxed-Co` have little effect and their results are not reported. The interference was one to four CPU hogs. Since SPECjbb performs little synchronization and `ab` had no synchronization, their CPU utilizations can achieve the fair share and IRS does not improve utilization as it did for the parallel workloads above. However, as shown in Figure 4.8, IRS was still able to improve the throughput of *SPECjbb* by up to 12%, though did not help much in `ab` (by as much as 4%). While IRS did not help increase utilization, it did improve request latency, which contributed to throughput improvement. Figure 4.8 (b) shows that the average latency of the new order transaction in *SPECjbb* was improved by as much as 46%.

In contrast, IRS had marginal improvement on `ab` latency. Figure 4.8(b) shows that there was only slight improvement on the tail latency (99^{th} percentile) of `ab`. The difference between `ab` and *SPECjbb* is that `ab` had many more threads than the number of vCPUs and each request was short. Since Linux is able to sense the contention at the hypervisor by dynamically updating the `rt_avg` load on each vCPU, the load balancer in the guest OS was able to distribute threads on vCPUs based on the level of interference experienced by each vCPU. Therefore, IRS can only help the thread that was running when its host vCPU was preempted. As ab had a large number of threads, improvement on a few threads did not contribute to the overall throughput but helped the tail latency to some extent.

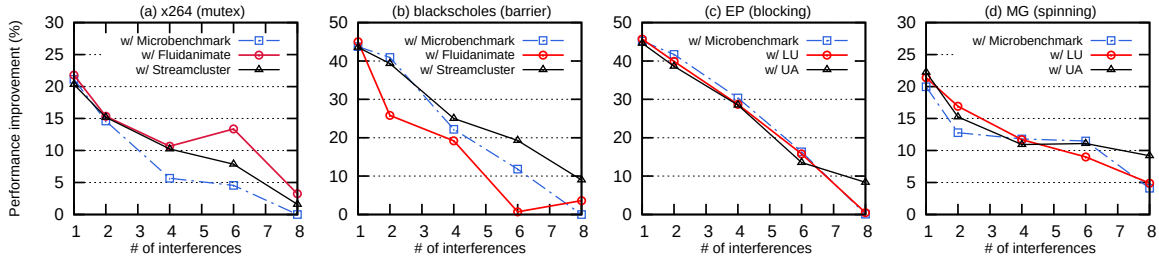


Figure 4.10: The trend of IRS performance improvement with a varying number of interferences.

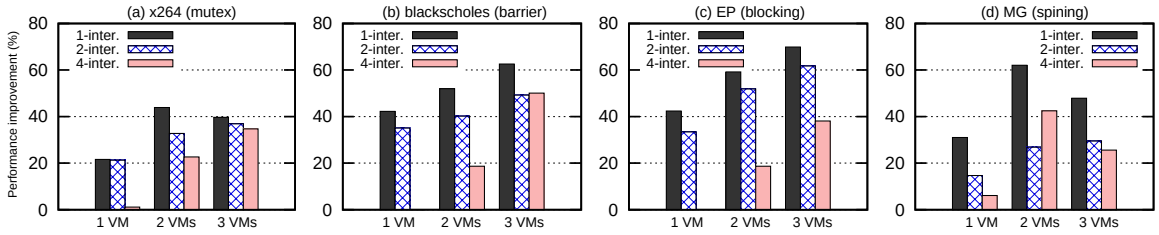


Figure 4.11: The trend of IRS performance improvement with a varying degree of interferences.

4.5.4 System Fairness and Efficiency

The objective of IRS is to allow the guest OS to more efficiently utilize its CPU allocation via enhanced in-guest load balancing. Since it requires some changes to both the guest OS and the hypervisor, we are interested in studying the system-wide fairness and efficiency when multiple realistic applications are co-located. The foreground VM ran various parallel workloads and had IRS enabled. We selected representative parallel programs as the interfering workloads running in the background VM. The interfering VM ran a vanilla Linux kernel and thus IRS had no effect on it [1]. We define the speedup of an application as its performance under IRS normalized to the performance in vanilla Xen/Linux. We use the weighted (average) speedup of the foreground and background applications to measure the overall system efficiency. The higher the weighted speedup, the higher system efficiency. A weighted speedup of 1 indicates the same performance as vanilla Xen/Linux. The foreground and background workloads were both repeated at least five times to ensure their execution completely overlapped with each other.

Due to space limits, we briefly report the fairness between the foreground and background VMs. IRS did not compromise fairness and the two VMs had a fair share of the pCPUs. The only change IRS made to the scheduling algorithm at the hypervisor is

[1] Without implementing the `VIRQ_SA_UPCALL` interrupt, the background VM ignores the SA notification sent by the hypervisor.

the delay added to each vCPU preemption for the guest OS to process SA notifications. Experimental results show that IRS improved the utilization of the foreground VM but the CPU consumption never exceeded the fair share.

Figures 4.7 and 4.9 show the weighted speedup for PARSEC and NPB benchmarks due to different scheduling strategies with a varying degree of interference. The weighted speedup follows the same trends of performance improvement in Figure 4.5 and 4.6. For PARSEC benchmarks (as shown in Figure 4.7), IRS had marginal or no speedup in `dedup` and `ferret`. For other workloads, IRS improved the system-wide speedup by as much as 40% and the average speedup across all workloads was 18% and 22% when the background workloads were `fluidanimate` and `streamcluster`, respectively.

An examination of the performance of foreground and background workloads revealed that the gain on system weighted speedup was mainly due to the performance improvement in foreground applications. In most cases, the background application had speedup in the range of -5% to 6%, with an exception for the case in which `raytrace` and `fluidanimate` were co-located and `fluidanimate` had 27% improvement. The performance degradation of the background application (as much as 5%) was due to the improved utilization of the foreground application. Thus, the background application had less CPU allocations. These results suggest that IRS did not change the way the background VM was scheduled by the hypervisor and most performance improvement of the foreground VM was due to more efficient load balancing in the guest OS. IRS never degraded the background performance significantly but had unexpected improvement for some background workloads, e.g., `fluidanimate` when running with `raytrace`.

Compared to IRS, `PLE` either had marginal improvement on the weighted speedup or hurt the overall system efficiency. For example, `PLE` degraded the weighted speedup considerably for `vips`, `bodytrack`, and `facesim`. Note that both the foreground and background VM had `PLE` enabled. The frequent trap into the hypervisor and the lack of coordination between the VMs were the culprits of degraded system efficiency. `Relaxed-Co` achieved better performance than `PLE`, but still hurt overall system efficiency when running `bodytrack` and `facesim`.

Similar results can be observed in Figure 4.9. For example, IRS improved system speedup for most application combinations except the `SP+UA` and `UA+UA` experiments. For spinning workloads, `PLE` and `relaxed-Co` had better worst-case results. For all experiments including those in Figure 4.7, IRS had no significant impact on system speedup in the *4-inter.* cases. The degradation or improvement of weighted speedup was within the range of -5% to 5%.

4.5.5 Scalability and Sensitivity Analysis

In this section, we extend the evaluation to a larger number of vCPUs per VM and consolidating more VMs. We are interested in quantitatively measure the effectiveness of IRS for different types of parallel workloads. All results were the average of five runs.

First, we created two 8-vCPU VMs and configured them to share 8 pCPUs. The foreground VM ran 8-thread parallel workloads while the background VM executed three different types of interferences: one CPU-bound synthetic workload and two real parallel applications. The level of interference varied, starting from one vCPU with interference to all vCPUs (8-vCPU) with interference. We selected four benchmarks from PARSEC and NPB for evaluation. `X264` exclusively uses `pthread mutexes` for point-to-point synchronization and `blackscholes` uses `pthread barriers` for group synchronization between threads. NPB benchmarks employ a data-parallel programming model and use barrier-like synchronizations. `EP` performs less synchronization and uses blocking synchronization. We set `MG` to use spinning synchronization.

Figure 4.10 shows the trends of performance improvement due to IRS relative to vanilla Xen/Linux. We have the following observations: (1) performance gain diminishes as the number of vCPUs having interference increased. When all vCPUs are experiencing interference, the average gain is marginal at about 4%. (2) Parallel workloads with different types of synchronizations respond differently to IRS. Programs with group synchronization, such as barriers, suffer more from LHP and LWP, thereby benefiting more from IRS. The performance gain of point-to-point synchronizations, e.g., mutexes, is less than that of group synchronizations. IRS is more effective for mitigating LHP and LWP problems in blocking synchronization than in spinning synchronization. (3) overall, the stated trends apply to all three types of interfering workloads.

Next, we fixed the number of vCPUs in the foreground VM to 4 and varied the number of interfering VMs from 1 to 3. For example, *1-inter.* with three interfering VMs refers to the case that one vCPU of the foreground VM has interference and there are 3 VMs competing for the CPU cycles on the same pCPU. Figure 4.11 shows that as the degree of interference increases on each interfered foreground vCPU, the performance gain of IRS increases in most cases. Another important observation was that IRS has significant improvement under high degree of interference even all vCPUs of the parallel VM experience interference (i.e., *4-inter. + 3VMs*). We conclude that IRS can be more useful in a highly consolidated scenario with many VMs sharing the same pCPUs.

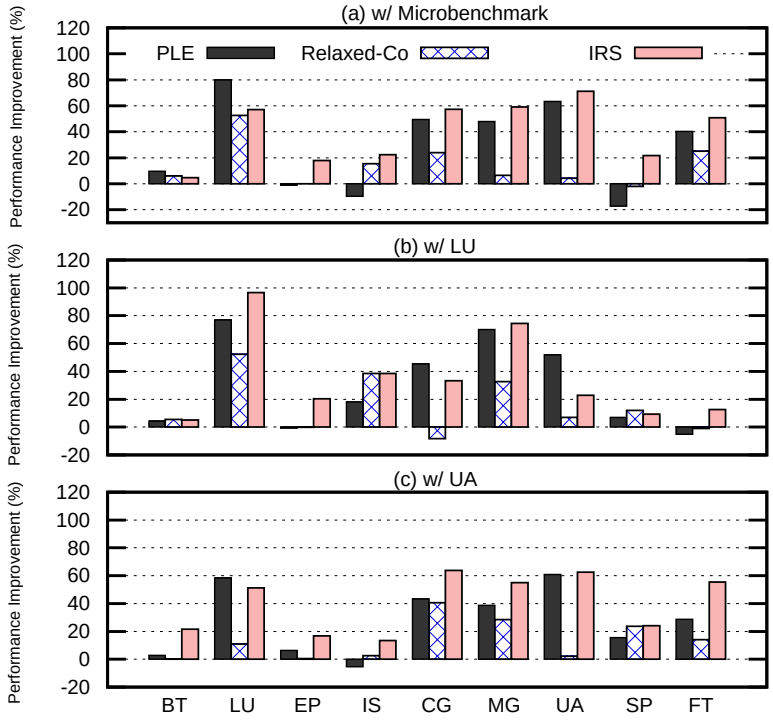


Figure 4.12: NPB performance in response to CPU stacking.

4.5.6 Mitigating CPU Stacking

We found that existing SMP schedulers, including the native Linux process scheduler and Xen's hypervisor-level vCPU scheduler, suffer from a severe CPU stacking problem when parallel workloads with frequent blocking are co-located with applications with persistent CPU demands. For example, if a four-thread parallel workload with blocking synchronization, e.g., `streamcluster`, shares 4 CPUs with three persistent CPU hogs, the parallel threads or vCPUs running these threads will be stacked on a single or a small number of CPUs, leaving much of the hardware-level parallelism unexploited by the parallel program. According to our PARSEC experiments with Linux CFS and Xen's credit scheduler, the four parallel threads were stacked on one or two cores and had 5-20x slowdown compared to the case in which threads or vCPUs were pinned to separate cores.

Root causes of CPU stacking Modern SMP schedulers are designed for scalability and proportional fair sharing. Each CPU in an SMP system runs an independent fair-sharing scheduler and relies on thread/vCPU migration for load balancing. The objective of load balancing is to evenly distribute workload onto multiple CPUs. and oftentimes the level of load is measured by the CPU utilization of a thread/vCPU. CPU stacking occurs if threads of a parallel program are placed on the same CPU and multiplexed in a time-

sharing manner. As a result, the stacked threads cannot execute simultaneously, leading to the loss of parallelism. CPU stacking occurs due to two reasons. **First**, Thread or vCPU scheduling is oblivious of the dependencies between parallel threads or vCPUs. Thus, placing sibling threads/vCPUs on the same CPU is legitimate as long as it satisfies fair sharing and load balancing. **Second and most importantly**, there exists a deficiency in existing SMP schedulers when scheduling blocking workloads. Due to LHP and LWP, parallel threads frequently block and wait to enter the critical section. Since blocked threads do not consume any CPU cycles, they exhibit *deceptive idleness* (DI) to the scheduler. This situation is similar to DI in disk scheduling [47] and causes blocking threads to be consolidated on a small number of CPUs due to their low CPU utilization.

Figure 4.12 shows the performance of NPB benchmarks under `PLE`, `relaxed-Co`, and IRS when all the vCPUs of the foreground and background VMs were unpinned. Performance is normalized to that in vanilla Linux/Xen and the interference was *4-inter.* CPU hogs. Since NPB benchmarks never block, the DI situation does not occur. As shown in Figure 4.12, all strategies were effective in improving NPB performance over the baseline and the degree of improve was significantly higher than that in Figure 4.6, indicating a mitigation of the CPU stacking problem. Among these scheduling strategies, `PLE` prevented excessive spinning and `relaxed-Co` balanced the progress of sibling vCPUs, thereby helping spreading them onto separate cores. Compared to `PLE` and `relaxed-Co`, IRS achieved overall higher performance gain, showing that in-guest load balancing is more resilient to CPU stacking caused by oblivious vCPU scheduling.

Figure 4.13 shows the performance of PARSEC benchmarks. Note that the stacking of PARSEC application threads was due to deceptive idleness caused by LHP or LWP. The figure shows that neither `PLE` nor `relaxed-Co` was generally effective in alleviating CPU stacking but exacerbated the performance slowdown. For example, `PLE` incurred up to 78% performance degradation compared to the baseline in `dedup`. Because the CPU stacking of blocking workloads is due to DI, `PLE`, which stops spinning vCPU and yields to competing vCPUs, caused more idling of the PARSEC workload. `Relaxed-Co` also caused considerable slowdown in some cases, e.g., `dedup`, `vips`, and `canneal`, because it only switched the leading vCPU and the lagging vCPU and was unable to address the stacking problem. In contrast, IRS proactively pushes threads from preempted vCPUs to idle or less loaded vCPUs, preventing these vCPU from idling. As discussed in § 4.5.2, With the help of IRS, blocking workloads avoided unnecessary idling and exhibited their factual CPU demand to the SMP scheduler. This helped prevent the DI problem and the resulted CPU stacking.

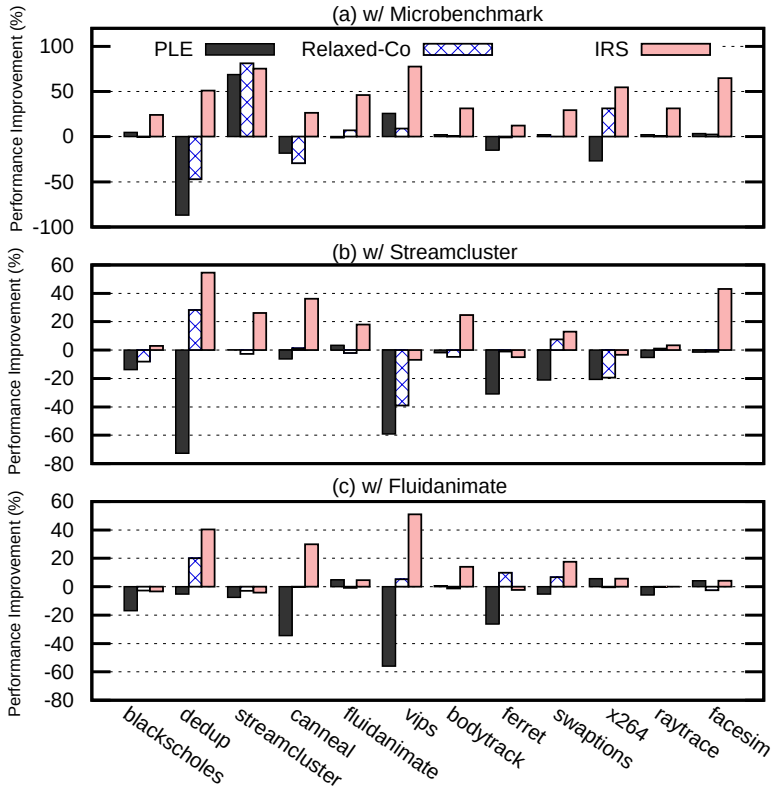


Figure 4.13: PARSEC performance in response to CPU stacking.

4.6 Summary

Limitation IRS proactively migrates preempted threads to another vCPU based on the estimation of load on the target vCPU. It cannot eliminate all vCPU idle time or achieve perfect load balancing because the load estimate can be inaccurate. The ideal migration should be pull-based and happen when a vCPU becomes idle. This calls for a new mechanism of task migration – migrating a "running" task from a preempted vCPU.

Autonomous guest resource management IRS allows the guest to migrate tasks among its sibling vCPUs and leads to more efficient utilization of its CPU allocation. Although small changes are needed in the guest kernel and hypervisor, we have shown that the changes do not affect the core resource scheduling algorithms at the hypervisor. We believe that hypervisors should provide such interfaces to the guest OS for autonomous and efficient guest resource management.

This chapter demonstrates that the semantic gap between the guest OS and hypervisor leaves the potential of addressing the LHP and LWP problems in the guest unexploited. We design IRS, a simple approach based the classical concept of scheduler activations to bridging the semantic gap and enhancing in-guest load balancing. Experimental results

show that IRS is especially effective for workloads that have a portion of threads with interference in a highly consolidated environment.

CHAPTER 5

Preemptive Multiple Queue Fair Queuing

5.1 Introduction

The prevalence of shared services, such as multi-tenant clouds [4, 38, 117], shared storage [48, 90], and multi-user clusters [27, 66], has led to a plethora of studies on fair and predictable resource allocation of shared resources. An important method for achieving resource fairness is using a fair queuing (FQ) scheduler, which allows competing tenants to take turns to use a shared resource. While FQ is a packet scheduling technique originally designed for sharing a network link between multiple flows, it has since been extended to managing various types of resources [90, 108], scheduling flows with variable packet lengths [11, 12, 28, 58], and supporting fair queuing on multiple links [14].

Modern operating systems (OSes) and hypervisors employ variants of FQ algorithms in the thread scheduler or the virtual CPU (vCPU) scheduler. In this context, an FQ scheduler allocates processor bandwidth, i.e., CPU cycles, to competing threads [1]. The time quantum each thread receives in a round corresponds to a packet serviced in the original FQ algorithm. On a single-core system, FQ schedulers effectively guarantee fair CPU allocation among active threads while allowing some threads to use more than their share if otherwise the core would become idle. This ensures that the scheduler is *work conserving*. However, on multicore systems, existing FQ schedulers fail to provide necessary resource isolation between competing applications, causing unfairness and performance unpredictability.

Figure 5.1 demonstrates the severity of unfairness in state-of-the-art multicore schedulers. We placed two multi-threaded applications to share the same set of cores, with the number of threads in each application matching the number of cores. We measured the aggregate CPU allocation to each application as a whole to evaluate the fairness of a multiprocessor scheduler. In the tests of hypervisor schedulers, the two applications ran in separate virtual machines (VMs) with the same number of vCPUs. Ideally, if *max-min* fairness is enforced, each application should receive a fair share of the total capacity of all available cores if the aggregate demand of all its threads exceeds the fair share. We empirically confirmed that both applications were able to consume almost all CPU when

[1] We use threads and vCPUs interchangeably.

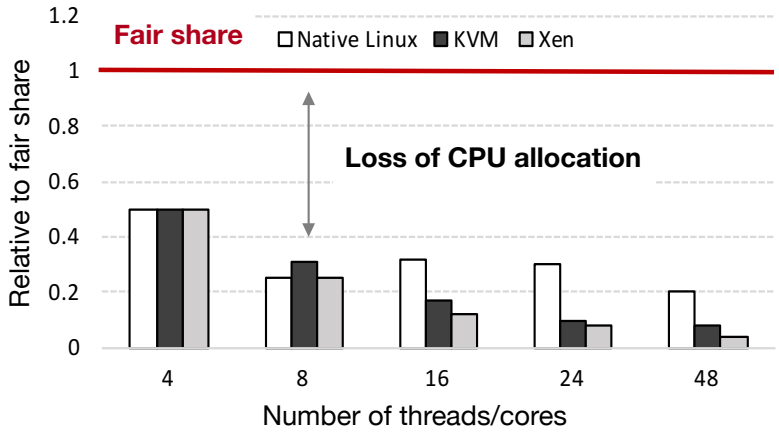


Figure 5.1: The deficiency in state-of-the-art multiprocessor schedulers. Multithreaded programs with blocking synchronization suffer unfair CPU allocation.

running in solo. The program under test was *streamcluster* from the PARSEC benchmark suite [103]. It employs blocking synchronization, which puts threads to sleep if they fail to enter the critical section. The collocated application was an arbitrary program with persistent CPU demand. It could be either a parallel program with busy-waiting (spinning) synchronization or a multiprogramming workload. As shown in Figure 5.1, *streamcluster* suffered unfair CPU allocation under all three multicore schedulers and the unfairness was aggravated as the number of threads/cores increased.

This previously unknown deficiency of multicore schedulers can cause significant performance slowdown and high variability to parallel applications in multi-tenant systems. As a consequence, due to concerns of poor service quality, leading public cloud providers, such as Amazon AWS, Microsoft Azure and Google Compute Engine, do not allow CPU multiplexing among symmetric multiprocessing (SMP) VMs. Such a conservative strategy diminishes the benefits of workload consolidation, resulting in low CPU utilization and high user cost. This deficiency can also hamper the adoption of the emerging serverless computing model, in which thousands of short-lived, possibly inter-dependent and/or chained containers are multiplexed on multicore systems.

The discovered unfairness in multicore scheduling is the result of the complex interplay between parallel workloads and OS thread schedulers. On the one hand, parallel programs rely on simultaneous access to CPU to make collective progress among multiple threads and otherwise suffer substantial performance slowdown if critical threads holding important locks are preempted. The remaining threads who are waiting on the synchronization cannot make progress, either performing futile spinning or being put to sleep (block). On the other hand, multicore schedulers enforce fair CPU allocation on a per-core basis

and are usually work-conserving. Therefore, threads that are idling due to synchronization forfeit their CPU shares, leading to unfair allocation between a parallel program and other competing programs. OS Load balancing could further aggravate this problem. Frequently idling threads, which show low CPU load, are gradually moved onto a few cores as consolidating fragmented load helps improve load balance. If CPU stacking occurs, sibling threads belonging to the same application compete with each other, introducing more idleness.

The culprit of such harmful interactions is twofold – 1) parallel programs exhibit *deceptive idleness* (DI) under contention, failing to expose their actual CPU demand to the OS scheduler; 2) there lacks a mechanism to fairly schedule inter-dependent threads on multiple CPUs. Therefore, parallel programs are unfairly penalized for being idling but not appropriately compensated. To address these issues, we extend the FQ algorithm for sharing a single network link to thread scheduling on multiple cores. We propose preemptive multi-queue fair queuing (P-MQFQ), a close approximation of the idealized generalized processor sharing (GPS) service discipline for multiple CPUs. P-MQFQ assumes a centralized queue to dispatch threads to multiple CPUs such that competing programs as a whole receive a fair share of the aggregated capacity of multiple CPUs. To tackle deceptive idleness, P-MQFQ allows threads from under-served programs to preempt currently running threads from other programs. As such, programs experiencing deceptive idleness are temporarily prioritized to catch up with those who have exceeded their fair shares.

Maintaining a centralized thread queue imposes a major scalability bottleneck in multicore scheduling. To implement P-MQFQ in real systems, we augment state-of-the-art multicore schedulers that use independent local queues with three queue operations: MIGRATE, PPREEMPT and SWITCH to approximate global fairness on multiple queues. We have implemented P-MQFQ in Linux completely fair scheduler (CFS), the default scheduler for native Linux and KVM, and Xen's credit scheduler. Experimental results show that P-MQFQ effectively addresses deceptive idleness and improves fairness in multicore scheduling. Parallel programs with blocking synchronization are able to utilize close to, but never exceed their fair shares. Our results also show significant improvement over three representative multicore scheduling optimizations.

5.2 Background and Problem Analysis

Classical fair queuing (FQ) algorithms are concerned with sharing a single network link among a set of flows [81]. Each flow consists of a sequence of packets that need to be transmitted through the shared link. Generalized processor sharing (GPS) is a reference model for fair queuing disciplines. A GPS server operates at a fixed rate r and can transmit multiple flows simultaneously. A flow is backlogged at time t_1 if a positive amount of that

flow's requests is queued at time t_1. Then the idealized GPS model guarantees that during any interval $[t_1, t_2]$, in which the set of backlogged flows $\mathscr{F}$ is unchanged, each backlogged flow i receives a minimum service rate r_i according to its weight ϕ_i:

$$r_i = \frac{\phi_i}{\sum_{j \in \mathscr{F}} \phi_j} r.$$

To approximate GPS in realistic systems that can only transmit one flow at a time, a number of packet-based GPS (PGPS) approximations have been developed [11, 12, 28, 37, 39, 40, 48, 70, 81, 82, 89, 122, 123]. PGPS transmits packet by packet in its entirety and only serves one packet at a time. PGPS approximates bandwidth allocation in GPS by serving packets in the increasing order of their finish time F under GPS. PGPS uses a notion of *virtual time* to track the progress of packet transmission in GPS. Virtual time measures the number of bits has been transmitted on a per-flow basis. System virtual time and a flow's virtual time advance at a rate of $\frac{r}{\sum_{j \in \mathscr{F}} \phi_j}$ and $\frac{\phi_i}{\sum_{j \in \mathscr{F}} \phi_j} r$, respectively. If there is only one backlogged flow, its virtual time advances at the rate of server capacity r and is identical to wall-clock time. When multiple flows are being serviced, flow virtual time is slower than real time and reflects its progress under the idealized GPS discipline. System virtual time stops advancing when the server is idle.

Virtual time-based PGPS implementation assigns each packet k from flow i a virtual start tag S_i^k and a virtual finish tag F_i^k when the packet arrives. Assume that server capacity is normalized to 1 and all flow weights sum to 1. The finish tag of a packet can be calculated based on its start tag:

$$F_i^k = S_i^k + \frac{L_i^k}{\phi_i}, \tag{5.1}$$

where L_i^k is size of the packet. The start tag is the maximum of system virtual time $v(t)$ at packet arrival time t and the finish tag of the last packet from the same flow:

$$S_i^k = \max\{F_i^{k-1}, v(t)\}. \tag{5.2}$$

The update of start tag ensures that inactive flows would not lag arbitrarily behind active flows in virtual time such that they could penalize active flows for utilizing the bandwidth left idle by inactive flows.

5.2.1 Start-time Fair Queuing

One obstacle to implement virtual time-based PGPS is the computation of system virtual time $v(t)$, which requires the simulation of a bit-by-bit GPS server. This simulation is computationally expensive to perform at each packet dispatch. Start-time fair queuing

(SFQ) [39] dispatches packets in the increasing order of start tags instead of finish tags. Ties are broken arbitrarily. Further, $v(t)$ is defined as the start tag of the packet in service at time t. SFQ offers two advantages [39]: 1) the packet size does not need to be known a priori and SFQ is able to handle variable server rates; 2) the computation of $v(t)$ is inexpensive as it only requires to examine the packets in service.

5.2.2 Fair-share CPU Scheduling

Similar to fair queuing in shared network, fair-share CPU scheduling aims to fairly allocate CPU bandwidth to competing threads. The time quantum each thread receives each time it runs on CPU is equivalent to dispatching a packet from a flow. Due to the advantages we previously discussed, SFQ and its variants are widely adopted in fair-share CPU scheduling. The default Linux completely fair scheduler (CFS) implements SFQ for fair sharing each individual CPU. CFS maintains a per-thread *virtual runtime* (vruntime) for each thread and tracks the minimum vruntime on a CPU. A thread's vruntime is updated each time it finishes a time quantum and the advancement is calculated based on the length of the time quantum and the thread's weight. CFS schedules threads based on the increasing order of their vruntimes. The minimum vruntime is defined as the maximum of the current minimum vruntime and the vruntime of the current running (in service) thread. It is updated each time a thread finishes a time quantum. When a thread wakes up from idling, its vruntime is set to the maximum of the current minimum vruntime and its vruntime before sleep.

In a multiprocessor (or multicore) [2] system, the operating system (OS) runs multiple copies of the fair-share CPU scheduling algorithm, one on each CPU, and relies on load balancing to evenly distribute threads over CPUs. Ideally, if all threads are runnable (backlogged) all the time and there are an equal number of threads on each CPU, fairly allocating CPU on a per-CPU basis leads to global fairness. However, system load often fluctuates over time and threads need to be migrated across CPUs to balance the load. Load imbalance undermines global fairness among threads as threads on CPU with higher load receive less CPU than those on CPUs with less load even they have the same weight. The fundamental problem is that weight is only significant to threads on the same CPU and affects the allocation on a particular CPU.

Unfortunately, thread migration is expensive as it requires double run queue locking to move a thread from the source to the destination CPU and it also undermines cache locality. Therefore, load balancing is performed infrequently and largely based on heuristics.

[2] We use multiprocessor and multicore interchangeably throughout this chapter to refer to multiple CPU queues.

For example, Linux performs load balancing on two occasions: 1) when a core becomes idle, it pulls threads from the busiest CPU; 2) the OS periodically moves threads from the busiest CPU to the least loaded CPU. The busyness of a CPU is measured by the number of runnable threads during a specified period. The busyness measure decays over time with recently runnable threads weighing more than older threads.

5.2.3 Deceptive Idleness

Despite fair-share scheduling on individual CPUs, parallel programs are susceptible to unfair CPU allocation on multiprocessors. A critical component of parallel programs is synchronization, which serializes the execution of threads in the critical section through locking. Threads could either spin on the lock or block if they failed to acquire the lock. Since futile spinning wastes CPU cycles and can inflict priority inversions [15], blocking and the hybrid spin-then-block synchronization, which eventually puts lock waiter threads to sleep, is widely adopted in parallel libraries. For example, the default implementations of mutex locks, barrier and semaphore in Pthread use blocking. Similar to an issue in disk scheduling [47], parallel threads using blocking synchronization exhibit *deceptive idleness* (DI) when a thread on the critical path is preempted. The critical thread could be a lock holder or a designated waiter to acquire the lock, which refers to the well-studied lock-holder preemption (LHP) [36] and lock-waiter preemption (LWP) problems [3, 79], respectively. However, it is not well understood why these problems cause cascading performance degradation.

Recall that SFQ-based fair sharing is work-conserving and does not penalize threads that consume resources that are otherwise left idle. Since system virtual time $v(t)$ on each CPU advances according to the start tag of the current running thread, the finish tag F_i of an idle thread is guaranteed to be smaller than $v(t)$ when it wakes up. Therefore, according to equation 5.2, an idle thread will align its start tag with $v(t)$ when waking up. This will allow the continuously running thread, which consumes more than its fair share, an equal opportunity to compete with the waking thread. As this pattern repeats, a frequently idling thread forfeits its share but is not compensated. Since in multiprocessor scheduling, CPUs independently enforce fair allocation on local queues, deceptive idleness costs parallel programs a significant proportion of their share of CPU. The heuristic-based load balancing further aggravates the issue. A CPU with deceptively idling threads appears to be lightly loaded and pulls threads from heavily loaded CPUs. If sibling threads belonging to a parallel program are stacked on the same CPU, intra-program CPU competition leads to severe serialization and more idleness.

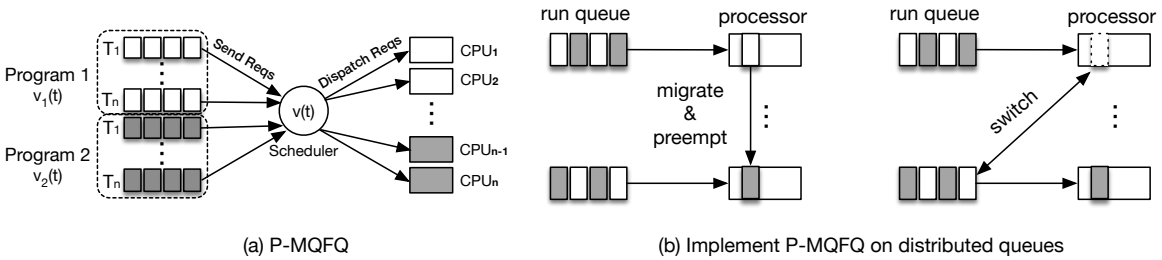
5.2.4 Multi-Queue Fair Queuing

Multi-queue fair queuing is concerned with sharing the aggregated capacity of multiple links among flows [14, 48]. A notable work extends SFQ to dispatch concurrent requests to utilize multiple links and controls the number of requests dispatched from different flows to enforce fair sharing [48]. The core problem is how to define system virtual time $v(t)$. Assume that there are D queues. Min-SFQ(D) defines $v(t)$ as the minimum start tag of any outstanding requests, which include queued and dispatched yet completed requests. SFQ(D) defines $v(t)$ as the maximum start tag of any dispatched yet completed requests.

The distinction between the two algorithms is important to addressing deceptive idleness. 1) Min-SFQ(D) advances $v(t)$ according to the lagging flow that cannot fully utilize its share. For example, a slow flow with no concurrency but always backlogged determines $v(t)$. Thus, it always has precedence over faster flows that use excessive resources. However, Min-SFQ(D) can cause starvation to fast flows if the slow flow issues a burst of requests. 2) SFQ(D) advances $v(t)$ according to the start tag of last dispatched request in backlogged flow. Therefore, it allows multiple queues to be fully utilized without penalizing backlogged flows or compensating lagging flows. Four-tag start-time fair queuing (FSFQ(D)) [48] combines the benefits of Min-SFQ(D) and SFQ(D) by maintaining four tags, the adjusted start and adjusted finish tags in Min-SFQ(D) and start and finisht tags in SFQ(D), for each request. Request scheduling is still based on start tags under SFQ(D) but ties are broken according to adjusted start tags under Min-SFQ(D). This compensates a lagging flow by giving it precedence in breaking ties.

Unfortunately, none of these algorithms is able to address deceptive idleness in parallel programs. Assume D queues (CPUs). When deceptive idleness occurs and there is only one critical thread active in a parallel program, in the worst case, there could be $D-1$ threads dispatched from an other backlogged program that are ahead of the blocked threads in the parallel program. Therefore, at each critical section of length l_c, a parallel program loses $(D-1)l_{max}$ utilization to another backlogged program after finishes Dl_{nc} cycles, where l_{max} is the time quantum and l_{nc} is the length of the non-critical section. Under Min-SFQ(D), which always treats the parallel program as a lagging flow, the parallel program receives at best $Dl_{nc}+l_{max}$ cycles on D CPUs in each round while the competing backlogged program receives $(D-1)l_{max}$ cycles. We use half of the total cycles allocated to both programs as the fair share and measure fairness using the *absolute relative lag* $|\frac{S_{\text{fair}}-S_{\text{parallel}}}{S_{\text{fair}}}|$, where S_{parallel} is the parallel program's CPU allocation. Therefore, we have

$$lag = |\frac{l_{max}-l_{nc}}{l_{max}+l_{nc}} - \frac{2l_{max}}{D(l_{max}+l_{nc})}|. \tag{5.3}$$



(a) P-MQFQ

(b) Implement P-MQFQ on distributed queues

Figure 5.2: (a) The idealized P-MQFQ model with a centralized request queue. (b) A practical implementation of P-MQFQ on distributed queues by augmenting multicore schedulers with three run-queue operations.

One can observe that when D is large and $l_{nc} << l_{max}$, $lag \rightarrow 1$. It suggests that parallel programs with fine-grained synchronization could suffer starvation on large-scale multicore systems due to deceptive idleness. Starvation is aggravated under SFQ(D) and FSFQ(D) as parallel threads forfeit CPU shares after they wake up.

5.3 Preemptive Multi-Queue Fair Queuing

Our analysis finds that the keys to address deceptive idleness are 1) deriving a global dispatch order of threads on multiple CPUs so as to enforce fairness at the program level; 2) devising a multi-queue fair queuing algorithm that allows threads from a lagging program to be timely scheduled on CPU. In what follows, we elaborate on the design of preemptive multi-queue fair queuing (P-MQFQ) to meet these goals and present a practical implementation of (P-MQFQ) in state-of-the-art multicore schedulers.

Figure 5.2 (a) shows the P-MQFQ model for multiprocessor scheduling. The objective is to schedule requests from different programs (i.e., P1 and P2) such that the aggregated CPU time received by all threads of each program is proportional to program weights. If processors are available, programs with multiple threads can have multiple requests dispatched. A request represents the CPU demand from one thread. The request service time is either the maximum time quantum a thread can run or the actual runtime if the thread blocks before its time quantum expires. A thread immediately re-submits a request for another time quantum if its current quantum expires or waits until waking up if it was blocked. Similar to the existing MQFQ algorithms, P-MQFQ maintains a centralized request dispatch queue, where requests are scheduled in the increasing order of their start tags. The centralized queue also tracks system virtual time $v(t)$. The algorithm is defined as follows:

1. System virtual time $v(t)$ is defined as the maximum start tag of all requests in service. Per-program virtual time $v_i(t)$ is defined as the maximum start tag of requests in ser-

vice that belong to a particular program. Defining virtual time according to requests in service allows lagging programs to preempt threads from aggressive programs.

2. If a program is backlogged, i.e., there is at least one thread from the program having a request running on a CPU or queued, a request's start tag is the maximum of the program's last finish tag and the program's virtual time. Otherwise, i.e., the program has been idling or is newly launched, request start tag is aligned with the system virtual time.
3. On arrival, if a request's start tag is smaller than the system virtual time, it indicates that another program receives more service than this program. The newly arrived request preempts the request with the largest start tag in service. Note that the preempted request is guaranteed to be from a different program as only such requests can advance system virtual time beyond the new request's program virtual time.
4. After the request with the largest start tag in service is preempted, the system virtual time $v(t)$ is updated to the second largest start tag in service. The finish tag of the preempted request is $S_i^k + \frac{l_r}{\phi_i}$, where S_i^k is the start tag, l_r is the amount of time the request has been running on CPU before being preempted and ϕ_i is the weight of the program.

The use of two types of virtual time, i.e., system virtual time and per-program virtual time, allows P-MQFQ to track the amount of service received by each program and prioritize lagging programs. Most importantly, request preemption guarantees that threads blocked due to synchronization can have requests immediately scheduled after waking up if they belong to a lagging program. This prevents deceptive idleness from happening.

5.3.1 Approximating P-MQFQ on Distributed Queues

P-MQFQ relies on a centralized queue to derive a global notion of virtual time and requires tracking per-program virtual time. However, state-of-the-art multicore schedulers employ a distributed queue architecture because it scales well with a large number of CPUs. Each CPU maintains per-CPU virtual time and independently enforces fair sharing on local queues. To approximate P-MQFQ without requiring to maintain the system and per-program virtual time, we augment multicore schedulers with distributed queues with three run queue operations: MIGRATE, PPREEMPT and SWITCH.

With distributed queues, there lacks a notion of global system virtual time. Per-CPU system virtual time progresses independently. Therefore, request start tags on different queues do not reflect the amount of service received by requests; in other words, start tags are not comparable across queues. Recall that P-MQFQ preempts the in-service request with the largest start tag. It is equivalent to finding the last dispatched request from the

program that received the most service. We relax the requirement for finding the global maximum of all start tags. Instead, as shown by the left figure in Figure 5.2 (b), after a request is completed (i.e., the time quantum expires) a thread preempts any threads currently running on a different queue that received more service than the preempting thread. We also ensure that the preempted thread belong to a different program. Since the per-CPU system virtual time progresses as the request start tag increases, we consider the current running thread on a queue with the fastest virtual time progression to have received the most service. If such a thread is found on a queue, the thread with an expired time quantum is MIGRATED to the queue and PREEMPTS the running thread.

Per-program virtual time tracks the aggregate service time of all threads in a program on multiple queues. During synchronization, per-program virtual time advances slowly as only the critical thread is active. This ensures that the critical thread has a small start tag and is always timely scheduled to avoid lock holder or waiter preemption. However, tracking per-program virtual time across multiple queues will impose a significant scalability bottleneck. We approximate the effect of per-program virtual time by ensuring that the critical thread is timely scheduled. As shown by the right figure in Figure 5.2 (b), if a thread is blocked (dotted box) before its time quantum expires, it iterates over queues to look for a runnable (queued) sibling thread from the same program. If found, P-MQFQ will switch these two threads and allows the selected runnable thread to use the remaining time quantum left by the blocked thread. The SWITCH operation moves the two threads but retains the virtual time on the original queues. The selected runnable thread will inherit the virtual time of the blocked thread and vice versa. This ensures that thread switching does not undermine fairness and programs can continuously receive service during synchronization.

5.3.2 Implementation

Native Linux and KVM Since the completely fair scheduler (CFS) in Linux is an implementation of SFQ on a per-CPU basis, P-MQFQ naturally extends to CFS. In CFS, each CPU (queue) independently maintains per-queue system virtual time (minimum vruntime), which is the vruntime of the current running thread. Minimum vruntime is updated at each timer interrupt (by default every 1ms in Linux) with the vruntime of the current running thread. P-MQFQ tracks the progression of per-queue minimum vruntime in a 3ms time window and considers the queue with the largest vruntime advancement as the one receiving most service. When a thread is blocked, it scans all queues to look for a runnable thread with the same parent and switch to the thread. This method finds sibling threads belonging to the same program in native Linux and sibling vCPUs in KVM.

Xen In Xen's credit scheduler, CPU allocation is measured by *credits*. As a vCPU consumes CPU, credits are debited and the balance determines the vCPU's priority. vCPUs with non-negative credit balance are assigned with the normal `UNDER` priority while those with negative balance are given a lower `OVER` priority. Xen refills vCPUs' credits at the beginning of each accounting period (every 30ms). Each time a vCPU is allocated the amount of credits that lasts for a time quantum (30ms in Xen). If a vCPU cannot use up its credits in an accounting period, the unused credits are discarded, which is intended to prevent vCPUs from accumulating credits. Although the credit scheduler does not use the notion of virtual time, the consumption of credits is equivalent to the progression of virtual time. To implement P-MQFQ, we record the discarded credits on each CPU in each accounting period. The CPU with the least discarded credits is considered to have received the most service and the running vCPU on this CPU is preempted. The implementation of the SWITCH operation is similar to that in Linux.

5.4 Evaluation

In this section, we present an evaluation of P-MQFQ in both a bare-metal Linux environment and two representative virtualized environments (KVM and Xen). We first study the effectiveness of P-MQFQ in addressing deceptive idleness in parallel workloads with blocking synchronization (§ 5.4.2). We then show that P-MQFQ can significantly improve the performance of parallel workloads with different types of synchronization (§ 5.4.3). Finally, we study the overall system efficiency under P-MQFQ when multiple parallel applications are each scheduled by P-MQFQ (§ 5.4.4).

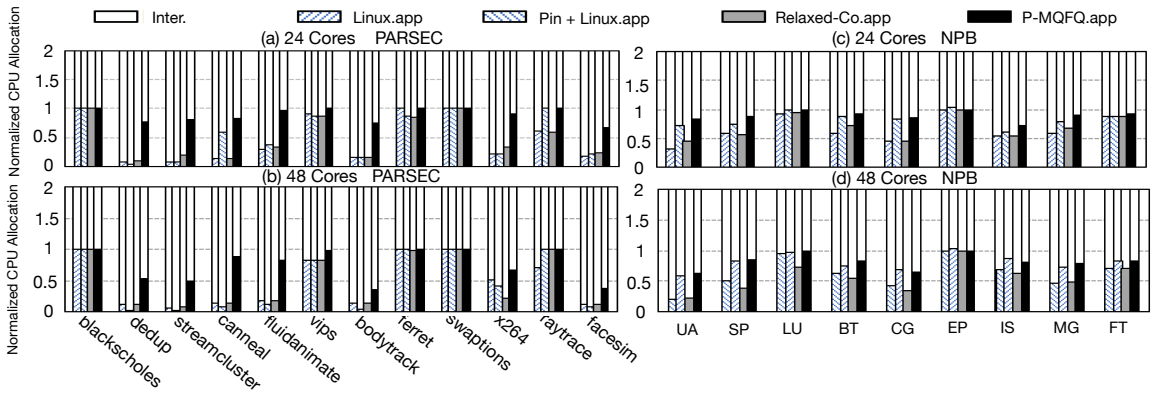


Figure 5.3: Normalized CPU allocation on physical machines for PARSEC (a, b) and NPB (c, d).

5.4.1 Experimental Setup

All experiments were performed on two servers. One is a DELL PowerEdge T420 server with two six-core Intel Xeon E5-2420 processors (24 cores with hyperthreading enabled) and 32GB memory. Another is a DELL PowerEdge R830 with four twelve-core Intel Xeon E5-4640 processors (48 cores with hyperthreading disabled) and 256GB memory. The settings were intended to demonstrate the effectiveness of P-MQFQ at different scales as well as with or without hyperthreading. Linux 4.1.39 was used as the native Linux OS, the host OS in KVM, and the guest OS in KVM and Xen. Xen 4.5.0 was used in the Xen test. For the tests in KVM and Xen, two VMs were used to run the parallel programs under test and the interfering workloads. All results were the average of 5 runs.

Workloads We selected the PARSEC [103] and NASA parallel benchmarks [8] as the parallel workloads under test. PARSEC is a shared memory parallel benchmark suite with various blocking synchronization primitives such as mutex locks, condition variables and barriers. We compiled PARSEC using pthreads and used the native input size. NASA parallel benchmarks include 9 parallel programs. We used the OpenMP implementation of benchmarks with the class C input size. We set the environment variable `OMP_WAIT_POLICY` to `INACTIVE` to enable blocking synchronization.

The background workloads include a micro-benchmark and two realistic applications. For example, the micro-benchmark consists of a number of CPU hogs that continuously competes for the CPU cycles and had almost zero memory footprint. Another two applications are `streamcluster` from PARSEC and `ua` from NPB with blocking synchronization at the granularity of 5-8ms and 250-500ms respectively.

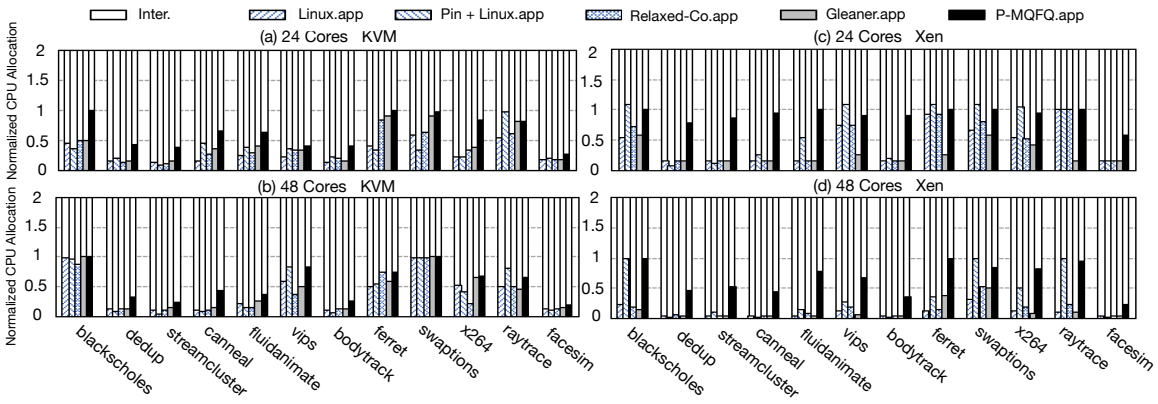


Figure 5.4: Normalized CPU allocation on KVM (a, b) and Xen (c, d) for PARSEC.

Scheduling strategies We compare the performance of P-MQFQ with the baseline multi-core schedulers and three representative scheduling strategies:

- `Linux CFS` and `Xen credit scheduler` were used as the baseline schedulers in native Linux, KVM and Xen, respectively.
- `Pin + {Linux, KVM and Xen}`: To avoid the severe CPU stacking issue due to deceptive idleness, we pinned threads and vCPUs to individual cores to disable OS load balancing. This strategy is often employed in production systems to improve performance predictability and preserve locality.
- `Relaxed-Co`: we implemented the VMware's relaxed co-scheduling in native Linux, KVM and Xen. Relaxed-Co monitors the execution skew of each vCPU (thread) and stops the vCPU that makes significantly more progress than the slowest vCPU. When a VM (program)'s leading vCPU (thread) is stopped, the hypervisor switches it with its slowest sibling vCPU to boost the lagging vCPU. Specifically, `Relaxed-Co` will monitor the execution skew of each thread or vCPU (scheduling entity) and stops the scheduling entity that makes significant more progress than the slowest one. A scheduling entity is considered to make progress when it executes instructions or it is on the `IDLE` state. Since VMware documentation does not reveal further details about relaxed co-scheduling, we implemented an optimization for parallel programs when a leading scheduling entity is stopped, the scheduler switches it with its slowest sibling to boost the execution of this lagging one.
- `Gleaner [31]`: In multi-tenant systems, CPU multiplexing causes suboptimal scheduling and fragmented CPU allocation in parallel programs. `Gleaner` consolidates fragmented CPU allocation into a few dedicated CPUs. Although CPU consolidation does not provide enough concurrency to user-level threads, it avoids expensive trapping to the hypervisor due to idling and harmful competition with co-running applications.

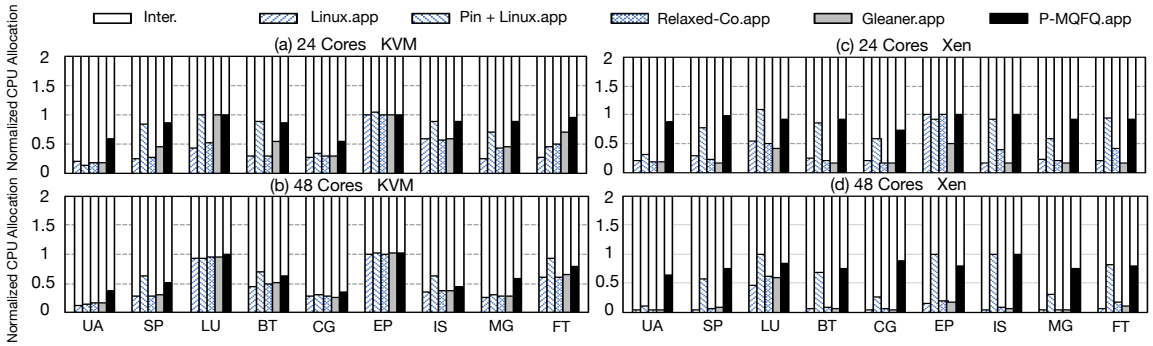


Figure 5.5: Normalized CPU allocation on KVM (a, b) and Xen (c, d) for NPB.

5.4.2 Addressing Deceptive Idleness

In this section, we evaluate the effectiveness of P-MQFQ in addressing deceptive idleness and improving the fairness of CPU allocation. Similar to the experiments in Figure 5.1, we collocated parallel workloads with a synthetic benchmark that had persistent CPU demand. The synthetic benchmark consisted of the same number of CPU hogs as the number of CPUs and was intended to create a contentious scenario to cause unfair CPU allocation. Usually, we say a scheduling algorithm is fair if and only if the amount of resource received by applications is proportional to their weights. Therefore, the ideal ratio of fair allocation will be 1:1 if two applications have the same weight value. In our evaluation, we report the fairness in two different levels: application or virtual machine level and thread or vCPU level. Regarding to the first one, parallel application were given the same weight to verify whether they receive the same amount of CPU resource. and improving performance for various parallel workloads on the physical machine. All experiments were first conducted on the machine with 24 cores (hyper-threading is enabled) and then on the machine with 48 cores to perform the scalability analysis (hyper-threading is disabled). Parallel applications from PARSEC and NPB were run with 24 and 48 threads respectively while background micro-benchmark was run with 23 and 47 threads. Figures 5.3, 5.4 and 5.5 show the normalized CPU allocation to the foreground parallel programs and the background interfering workload. The stacked bars show the allocation to the parallel applications (e.g., `P-MQFQ.app`) and the interfering workload (e.g., `Inter.`) under various approaches. A normalized allocation of 1 refers to fair allocation while a value less than 1 indicates that parallel programs receive less than the fair share. From these figures, we have the following observations:

First, compared with the baselines, P-MQFQ significantly increased the CPU allocation to most parallel workloads. P-MQFQ is most effective for programs with fine-grained synchronization. For example, among PARSEC benchmarks, *dedup*, *streamcluster* and *canneal* benefited most from P-MQFQ. According to the equation 5.3, the shorter the non-critical section (l_{nc}), i.e., more frequent synchronization, the higher degree of unfairness. In comparison, the improvement on CPU allocation was less in NPB benchmarks, which have much longer non-critical sections and less frequent synchronization. This observation was consistent both in the physical and virtualized environments.

Second, most benchmarks suffered more from deceptive idleness as the number of CPUs scaled but P-MQFQ also had diminishing gains. Intuitively, deceptive idleness is aggravated with a larger number of threads as more threads would be idling during synchronization. As shown in equation 5.3, the degree of unfairness increases with the number of CPUs D. P-MQFQ mitigated deceptive idleness by prioritizing the critical thread to avoid unnecessary idleness. However, P-MQFQ was unable to entirely eliminate idleness at

scale, in which the critical section weighed more to the non-critical section. We empirically confirmed that fine-grained programs, such as *streamcluster* and *canneal*, were unable to utilize their fair shares even in solo mode. For example, *streamcluster* only utilized around 1400% (equivalent to the capacity of 14 CPUs) CPU on the 48-core machine with 48 threads. This limits the gain of P-MQFQ at scale.

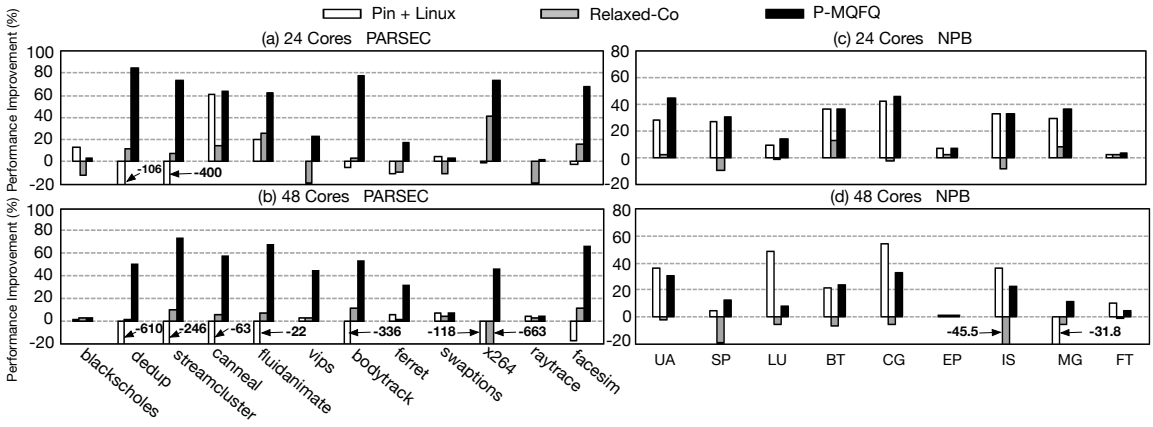


Figure 5.6: Performance improvement of PARSEC (a, b) and NPB (c, d) on physical machine.

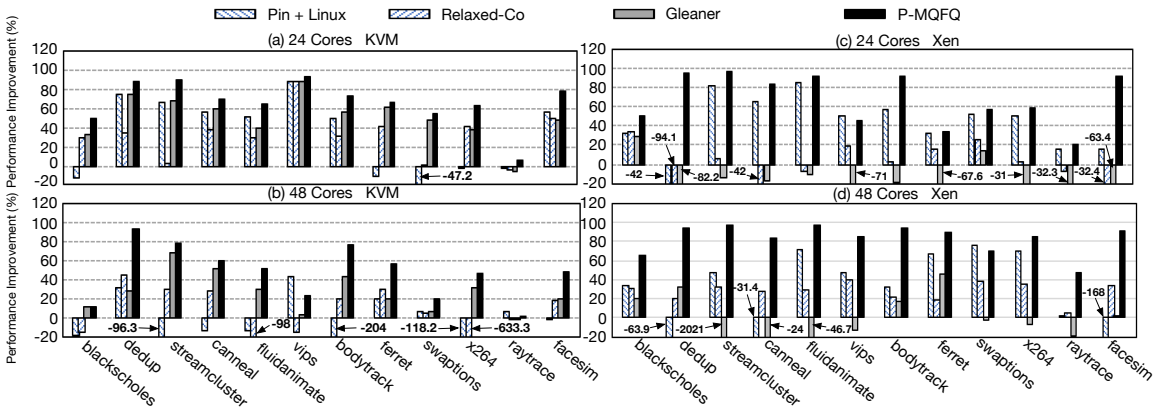


Figure 5.7: Performance improvement of PARSEC on KVM (a, b) and Xen (c, d).

Third, different multicore schedulers suffered differently from deceptive idleness, which also affected the effectiveness of P-MQFQ. Xen employs a longer time quantum l_{max} (i.e., 30ms) than that in CFS (i.e., 6ms). Therefore, parallel programs' CPU allocation

was much lower in baseline Xen compared to Linux and KVM. This does not affect the effectiveness of P-MQFQ as thread preemption in P-MQFQ effectively mitigate deceptive idleness, regardless of the length of the time quantum. P-MQFQ was able to achieve a similar level of utilization improvement in Xen compared to that in Linux. In contrast, P-MQFQ was less effective in KVM. Although both native Linux and KVM use CFS as the scheduler, in KVM, the scheduling entity is the vCPU. KVM employs hardware assisted virtualization (e.g., Intel VT-x) to virtualize vCPUs and thus incurs higher overhead for vCPUs migration. Significantly, a vCPU is not immediately eligible for migration after it is preempted because of a write barrier to enforce consistency across cores. In KVM, it takes more than 1ms before P-MQFQ can migrate a critical thread after the thread is preempted, thereby unable to timely schedule the critical thread. As a result, P-MQFQ achieved lower CPU allocation in KVM than that in Linux and Xen.

Fourth, P-MQFQ achieved a higher CPU allocation than the three representative scheduling strategies `Pin`, `Relaxed-co` and `Gleaner` in all tests except for *EP* from NPB, which is embarrassingly parallel and uses no synchronization. The key in P-MQFQ to increasing CPU utilization is to eliminate idleness as much as possible. To achieve this goal, P-MQFQ allows a preempted thread a chance to continue running by preempting another over-serving threads and moves threads across CPUs to avoid forfeiting allocated time quantum. Neither of the three approaches devise both optimizations. `Pin` avoids CPU stacking due to deceptive idleness but does not address the preemption of critical threads; `Relaxed-co` could not guarantee the simultaneous progress of all threads and thus is still susceptible to idleness-induced CPU stacking; thread consolidation in `Gleaner` does not expose enough parallelism to the user-level thread, thereby aggravating serialization at the critical section.

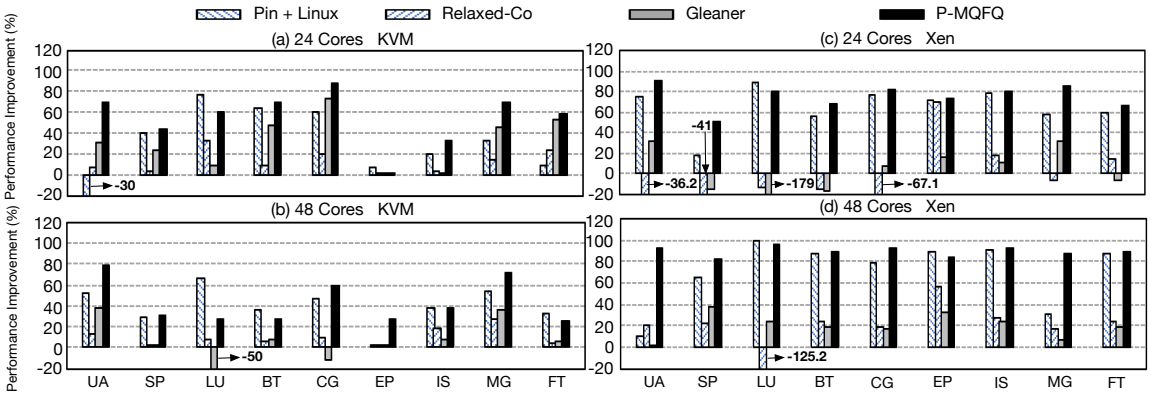


Figure 5.8: Performance improvement of NPB on KVM (a, b) and Xen (c, d).

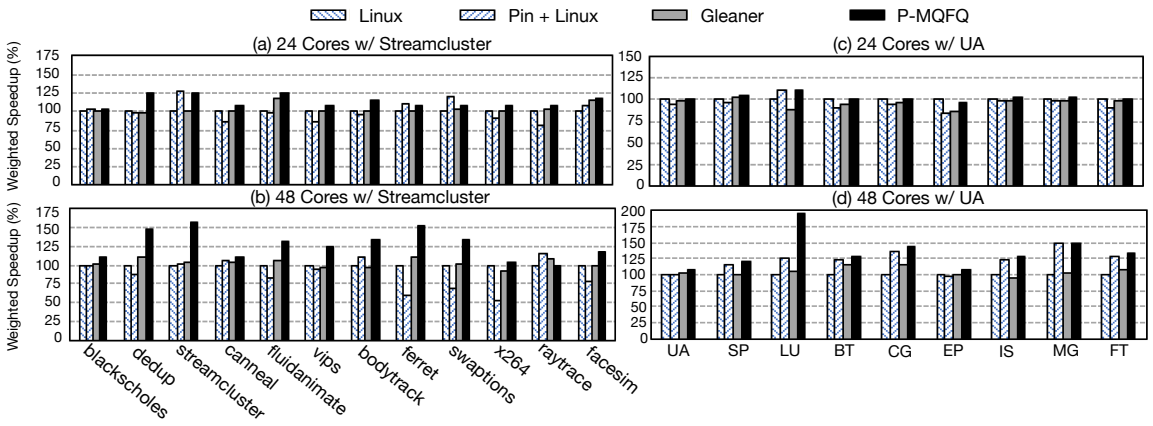


Figure 5.9: Weighted speedup of PARSEC (a, b) and NPB (c, d) on physical machine.

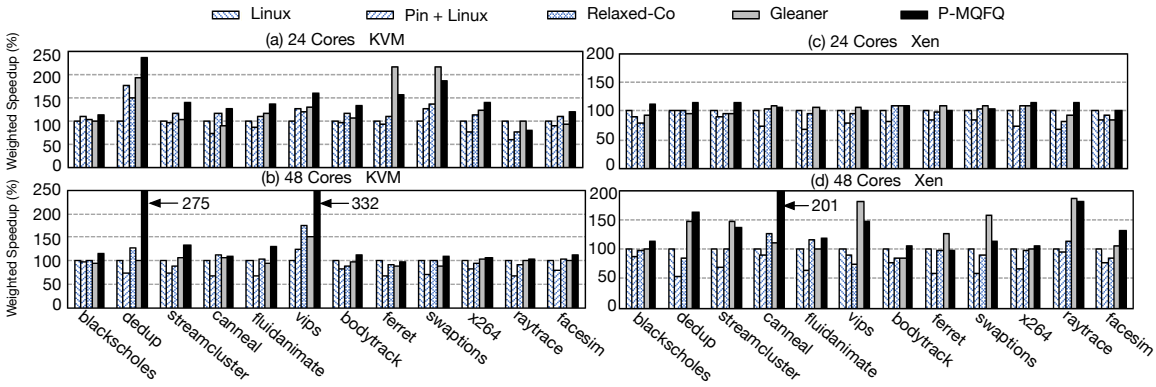


Figure 5.10: Weighted speedup of PARSEC and streamcluster on KVM (a, b) and Xen (c, d).

for workloads with fine-grained blocking synchronization about 5ms-7ms in PARSEC benchmarks, threads will be stacked on a single core when running on native Linux since CFS scheduler in multicore systems does not strictly follow the scheduling principle of SFQ(D) which will delay the backlogged threads with fast progress and reserve the CPU resource for the slowest threads. Another reason was that the minimum balancing interval in CFS will be 256ms. As such, CFS scheduler was unable to balance these stacking threads across different cores. Relaxed co-scheduling did not improve or even hurt the fair allocation compared to native Linux such as `bodytrack, x264`. The reason is that `Relaxed-Co` is specially designed to balance the progress of sibling threads through

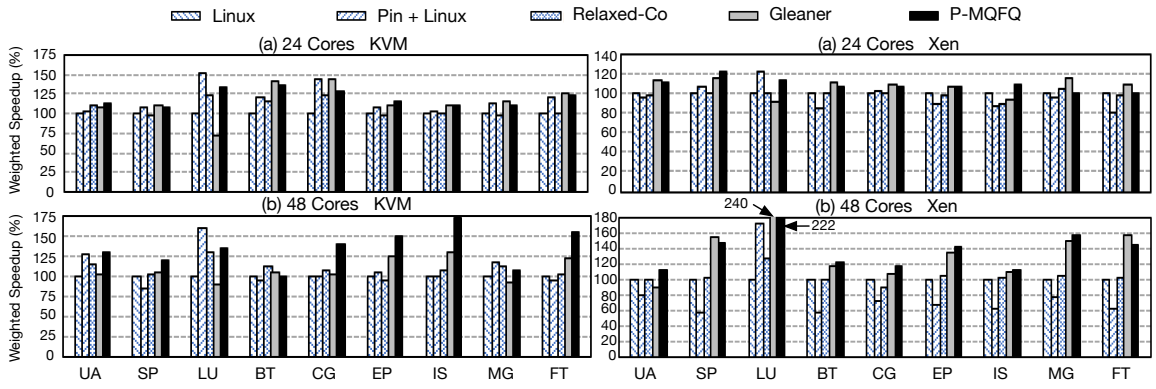


Figure 5.11: Weighted speedup of NPB and UA on KVM (a, b) and Xen (c, d).

switching the fast thread with slowest one. Unfortunately, the threads were still left in the stacking status without any change after doing this switch.

5.4.3 Improving Performance

For programs with blocking synchronization, scheduling inefficiencies manifest as excessive idleness. Thus, it is expected that P-MQFQ leads to performance improvement as it can eliminate much of the idleness and improve CPU utilization. Figures 5.6, 5.7 and 5.8 show the performance of PARSEC and NPB benchmarks due to different scheduling strategies in native Linux, KVM and Xen, respectively. Performance is normalized to that in the baselines, i.e., vanilla Linux, KVM and Xen. From these figures, we can see that P-MQFQ improved the performance of all benchmarks compared to that in the baselines. The performance improvement was up to 97% and 96% for PARSEC and NPB benchmarks, respectively.

In contrast, there were benchmarks suffering performance degradation under other approaches. For example, *streamcluster* had a degradation of 400% under `Pin` on the 24-core physical machine since `Pin` cannot mitigate the deceptive idleness; `Relaxed-co` degraded *x264* by as much as 663% on the 48-core physical machine due to the loss of 25% CPU allocation; `Gleaner` inflicted a 2021% slowdown to *streamcluster* on the 48-core Xen machine. The reason is that Xen hypervisor employed the comparison of vCPU priority to decide the load balance. Therefore, consolidating more threads into a single vCPU in Xen would make the stacking problem become more seriously and the performance would be further degraded.

5.4.4 System Fairness and Efficiency

The results presented so far focused on improving the utilization and performance of the foreground parallel programs. P-MQFQ did not take effect for the background workloads. We are also interested in evaluating P-MQFQ in managing multiple parallel workloads, each is actively scheduled by P-MQFQ. We collocated two blocking parallel workloads to share the same set of CPUs and used the geometric mean of individual programs' speedups (weighted speedup) to measure the overall system efficiency. The higher the weighted speedup, the higher the system efficiency. A weighted speedup of 1 indicates the same performance as the baseline system. The foreground and background workloads were both repeated at least five times to ensure their execution completely overlapped with each other. We selected *streamcluster* as the background workload for the PARSEC benchmarks and *UA* for the NBP benchmarks.

Figures 5.9, 5.10 and 5.11 show the weighted speedup of PARSEC and NPB benchmarks due to different scheduling strategies in native Linux, KVM and Xen, respectively. Our results indicated that P-MQFQ improved the system-wide weighted speedup by as much as 232%. In a physical environment, it achieved an average weighted speedup of 13% and 27.7% for PARSEC and NBP benchmarks, respectively. The overall performance improvement for both the foreground and background benchmarks were higher in virtualized environments (KVM and Xen). An examination of the performance of foreground and background workloads revealed that the gain on system-wide weighted speedup was due to the performance improvement in both applications. Furthermore, both foreground and background applications had improved CPU utilizations.

We also investigated the CPU allocation between these two applications and the result showed P-WQFQ could improve their CPU allocation to gradually close to the ideal ratio. For example, CPU allocation of `dedup` and `streamcluster` increased from 595%, 949% to 900% , 1080% respectively. Due to the space limits, the experimental results are not presented at here.

Compared to P-MQFQ, the pinning mechanism either had marginal improvement on the weighted speedup or hurt the overall system efficiency. For example, pinning degraded the weighted speedup considerably for `x264` on physical machine, `ferret` on KVM and `dedup` on Xen. `Relaxed-Co` achieved better performance than pinning, but still hurt overall system efficiency when running `streamcluster` and `facesim`. `Gleaner` performed better in some cases than P-MQFQ such as `ferret` and `x264` on 24 cores KVM. Overall, P-MQFQ significantly outperformed these approaches in the average speedup over all workloads combinations.

5.5 Summary

Spinning workloads face a different challenge in multi-tenant systems. Futile spinning due to the preemption of the critical thread wastes the fair CPU share of the spinning workload and can cause priority inversions. Since spinning workloads do not block, they do not suffer from deceptive idleness and thereby are unable to benefit from P-MQFQ. Recent advances in processor design allow the OS to detect excessive spinning through hardware-based techniques, such as pause-loop exiting (PLE), and to forcibly stop (blocks) a spinning thread. PLE is especially useful in virtualized environments, where the hypervisor is oblivious of spinning activities inside VMs. As such, spinning workloads will suffer deceptive idleness when threads are involuntarily put to sleep and thus they also can benefit from P-MQFQ.

Overhead Approximating a centralized request dispatch queue requires synchronization between CPUs. Our implementation of P-MQFQ on multicore schedulers incurs two types of overhead. First, to identify the thread that received most service, P-MQFQ needs to monitor the progression of virtual time on all CPUs and find the queue with the largest virtual time progression. This requires traversing all CPUs every 3ms. To isolate the overhead due to this operation, we compared program performance with and without P-MQFQ in solo mode, in which no thread migration is performed. We found approximating global virtual time incurs negligible overhead, adding an average of 1.3%, 3.7% and 2.4% overhead to the program execution time in Linux, KVM and Xen respectively.

Second, frequent thread migrations undermine cache locality and require an expensive run queue operation in multicore schedulers – double run queue locking, which locks the source and destination CPUs before a thread migration is completed. However, it is difficult to measure the overhead in scheduling parallel programs. When deceptive idleness occurs, P-MQFQ effectively improves CPU allocations to parallel programs, though inevitably incurs high overhead. It is important to study which factor, improving utilization or expensive thread migration, weighs more in overall performance. We consider that the overhead of thread migration dominates overall performance if P-MQFQ outperforms `Pin` in CPU utilizations but results in less performance improvement. For example, P-MQFQ outperformed `Pin` in CPU utilization (10% vs. 2.5%) but achieved less performance improvement than `Pin` (26.4% vs. 66.9%) in KVM. We compared the performance of P-MQFQ and `Pin` in all experiments, including those with the synthetic benchmark and with a realistic parallel workload, in Linux, KVM and Xen. In a physical environment, 44 out of 48 PARSEC tests and 33 out of 36 NPB tests show that the benefit of P-MQFQ outweighed its overhead. Similar observations were also made in KVM and Xen.

This chapter identifies an important deficiency in state-of-the-art multicore schedulers that causes unfair CPU allocation to parallel programs with blocking synchronization and leads to severe performance degradation. This deficiency hampers CPU multiplexing

chapterin shared services, such as public clouds. We attribute the deficiency to the inability of existing schedulers to deal with deceptive idleness and the lack of multi-queue fair queuing in the context of thread scheduling. To this end, we proposed preemptive multi-queue fair queuing (P-MQFQ), a centralized algorithm that uses thread preemption to guarantee fair CPU allocation for multi-threaded programs on multiple CPUs. P-MQFQ can be approximated by augmenting distributed queue-based schedulers with three run queue operations. Results show that P-MQFQ improves utilization and performance compared to three representative scheduling strategies.

CHAPTER 6

Conclusions and Future Work

This dissertation aims to fully understanding and optimizing the parallel performance in multi-tenant cloud. In this chapter, we summarize the approaches presented in this dissertation and give the directions for the future work.

6.1 Conclusions

As we know, cloud providers usually provide services in the form of giving customers the privileges to access the virtual machines. Symmetric Multiprocessing virtual machines (VMs) are thus gradually becoming increasingly common in cloud datacenters. They are often used by cloud users to host multi-threaded applications. On the other hand, cloud providers prefer oversubscribing their datacenters by consolidating multiple independent VMs onto a single machine to improve hardware utilization and reduce energy consumptions. However, CPU oversubscription introduces several challenges to efficiently executing parallel and multi-threaded programs in SMP VMs.

One of the challenges is to understand why the performance of parallel programs is notoriously difficult to reason about in virtualized environments. Although performance degradations caused by virtualization and interferences have been extensively studied, there still lacks a comprehensive understanding why parallel programs have unpredictable slowdowns when co-located with different types of workloads. We present a systematic study of parallel performance under interference. We find that the speed of individual threads under interference is determined by their varying resilience to interferences and the computation required to complete the parallel program can change vastly under interference due to alleviated intra-program contentions. Further, the overall performance is the result of the complex interplays between these factors. Avoiding harmful vCPU preemptions or maintaining asynchrony between vCPUs helps reduce slowdown under interference for different kinds of workloads. Inspired by these findings, we develop an accurate online approach for predicting slowdowns under interference without requiring completing the parallel program, and devise two scheduling optimizations at the hypervisor to improve performance

Locker holder preemption (LHP) and locker waiter preemption (LWP) are the two classic problems which are caused by the semantic gap inherently existing in the virtualized environment between the Guest OSes and hypervisors. In both two scenarios, the challenge is that hypervisors such as KVM and Xen are completely unaware of the activities in the guest OS and adversely deschedules virtual CPUs (vCPUs) that are executing in critical sections. This semantic gap will seriously degrade the performance of parallel applications in which they employ the synchronization primitives to protect the critical sections. We demonstrates the semantic gap between the Guest OS and hypervisor leaves the potential of addressing the LHP and LWP problems in the guest unexploited. We design IRS, a simple approach based the classical concept of scheduler activations to bridging the semantic gap and enhancing in-guest load balancing.Experimental results show that IRS is especially effective for work-loads that have a portion of threads with interference in a highly consolidated environment.

CPU schedulers, a key component in an OS design, has been under constant development for several decades. As new hardware emerges such as multicore processors, schedules have been adaptively scaled from singe core to multicore processor. Hypervisors in cloud, like schedulers in traditional OS, has also been developed to support the multiprocessor platforms. Two key objectives of schedulers and hypervisors are to fairly allocate the CPU resources between applications, users and VMs. However, we identifies an important deficiency in state-of-the-art multicore schedulers that causes unfair CPU allocation to parallel programs with blocking synchronization and leads to severe performance degradation. This deficiency will seriously hamper CPU multiplexing in shared services, such as public clouds. We attribute the deficiency to the inability of existing schedulers to deal with deceptive idleness and the lack of multi-queue fair queuing in the context of thread scheduling. To this end, we proposed preemptive multi-queue fair queuing (P-MQFQ), a centralized algorithm that uses thread preemption to guarantee fair CPU allocation for multi-threaded programs on multiple CPUs. P-MQFQ can be approximated by augmenting distributed queue-based schedulers with three run queue operations. Results show that P-MQFQ improves utilization and performance compared to three representative scheduling strategies.

6.2 Future Work

There are several issues and new research directions along with the line of this work. We have discussed that one key characteristic of the cloud is workloads in each VM will experience the interferences which include the resource contentions (inter-contention) on the hardware resources with other VMs and inter-thread contention such as lock contention

(intra-contention) between different threads inside the parallel applications. In this dissertation, we are mainly working to understand and optimize parallel performance in the virtualized environment from the perspective of inter-contention. Another way we could do such kind of performance analysis is from the intra-contention which inherently exists in parallel applications.

As we know, parallel applications usually employed the synchronization primitives such as mutex, spinlock and condition variables to implement the data sharing. Each thread will take turns to grab the lock in order to access the shared data. If all threads reside in the same NUMA node, there will be no serious cache coherence traffic during the program execution. However, things will become worse if the number of threads scale up to match the multicore multiprocessor NUMA architecture. The order each thread acquires the lock may cause significant difference to the volume of cache coherence traffic and thus affect the parallel performance.

REFERENCES

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, 2006.

[2] J. Ahn, C. H. Park, and J. Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proc of MICRO*, 2014.

[3] 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 IEEE/ACM International Symposium on Microarchitecture (Micro-47)*, 2014.

[4] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2.

[5] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1), 1992.

[6] Apache HTTP Server Benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html.

[7] A. C. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Trans. Comput. Syst.*, 19(3), 2001.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC'91)*, 1991.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, L. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, L. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of SOSP*, 2003.

[11] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithm. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'96)*, 1996.

[12] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case Fair Weighted Fair Queueing. In *Proceedings of 15th International Conference on Networking for Global Communications (INFOCOM'96)*, 1996.

[13] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4), 2010.

[14] J. M. Blanquer and B. Ozden. Fair Queueing for Aggregated Multiple Links. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'01)*, 2001.

[15] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The Convoy Phenomenon. *SIGOPS Operating System Review*, 13(2), 1979.

[16] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*, 2010.

[17] K. Chakraborty, P. M. Wells, and G. S. Sohi. Supporting overcommitted virtual machines through hardware spin detection. *IEEE Trans. Parallel Distrib. Syst.*, 23(2), 2012.

[18] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus Fair Scheduling: A Proportional-share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI'00)*, 2000.

[19] A. Chandra, M. Adler, and P. Shenoy. Deadline Fair Scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems. In *Proceedings of 7th IEEE Symposium on Real-time Technology and Applications (RTTAS'01)*, 2001.

[20] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. of HPCA*, 2005.

[21] L. Cheng, J. Rao, and F. C. Lau. vScale: Automatic and efficient processor scaling for smp virtual machines. In *Proc. of Eurosys*, 2016.

[22] L. Cheng, J. Rao, and F. C. M. Lau. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, 2016.

[23] Co-scheduling SMP VMs in VMware ESX Server. http://communities.vmware.com/docs/DOC-4960.

[24] T. Creech, A. Kotha, and R. Barua. Efficient multiprogramming for multicores with scaf. In *Proc. of MICRO-46*, 2013.

[25] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of ASPLOS*, 2013.

[26] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of ASPLOS*, 2014.

[27] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[28] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of Symposium on Communications Architecture and Protocals (SIGCOMM'89)*, 1989.

[29] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Proc. of ISPASS*, 2011.

[30] X. Ding, B. P. Gibbons, A. M. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proc. of USENIX ATC*, 2014.

[31] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the Block-waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*, 2014.

[32] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proc. of USENIX ATC*, 2014.

[33] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'99)*, 1999.

[34] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proc. of SIGMETRICS*, 1996.

[35] R. Essick. An Event-based Fair Share Scheduler. In *Proceedings of the Winter 1990 USENIX Conference*, 1990.

[36] T. Friebel and S. Biemueller. How to Deal With Lock Holder Preemption. In *Xen Developer Summit*, 2008.

[37] S. J. Golestani. A Self-clocked Fair Queueing Scheme for Broadband Applications. In *Proceedings of 13th International Conference on Networking for Global Communications (INFOCOM'94)*, 1994.

[38] Google Cloud Platform. http://cloud.google.com/compute.

[39] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Scheduling Networks. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'96)*, 1996.

[40] A. G. Greenberg and N. Madras. How Fair is Fair Queueing. *Journal of ACM*, 39, 1992.

[41] F. Guo and Y. Solihin. A framework for providing quality of service in chip multi-processors. In *Proc. of MICRO*, 2007.

[42] T. Harris, M. Mass, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proc. of EuroSys*, 2014.

[43] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63, 1984.

[44] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. December 2009.

[45] Intel TBB. http://software.intel.com/en-us/intel-tbb.

[46] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policy and architecture for cache/memory in cmp platforms. In *Proc. of SIGMETRICS*, 2007.

[47] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[48] W. Jin, J. S. Chase, and J. Kaur. Interposed Proportional Sharing for Storage Service Utility. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04)*, 2004.

[49] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of USENIX ATC*.

[50] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC'06)*, 2006.

[51] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proc. of SC*, 2012.

[52] S. Kashyap, C. Min, and T. Kim. Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSYS'15)*, 2015.

[53] J. Kay and P. Lauder. The Fair Share Scheduler. *Communications of the ACM*, 31, 1988.

[54] Kernel Based Virtual Machine. http://www.linux-kvm.org.

[55] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based coordinated scheduling for smp vms. In *Proc. of ASPLOS*, 2013.

[56] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based coordinated scheduling for smp vms. In *Proc. of ASPLOS*, 2013.

[57] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-Based Coordinated Scheduling for SMP VMs. In *Proceedings of 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, 2013.

[58] L. Kleinrock. *Queueing System*. 1975.

[59] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst.*, 15(1), 1997.

[60] T. Li, D. Baumberger, and S. Hahn. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round Robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Computing (PPoPP'09)*, 2009.

[61] F. Liu and Y. Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proc. of SIGMETRICS*, 2011.

[62] X. Liu and B. Wu. Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proc. of SC*, 2015.

[63] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, 2016.

[64] Z. Majo and T. Gross. Memory system performance in a numa multicore multiprocessor. In *Proc. of SYSTOR*, 2011.

[65] 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 *Proc. of MICRO*, 2011.

[66] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible collocations. In *Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (Micro-44)*, 2011.

[67] P. E. McKenney. Differential profiling. *Software - Practice and Experience*, 29(3), 1999.

[68] I. Molnar. Completely Fair Scheduler. In *Linux Journal*, 2009.

[69] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. of MICRO*, 2007.

[70] J. Nagle. On Packet Switches with Infinite Storage. In *RFC 970, FACC Palo Alto*, 1985.

[71] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, 1997.

[72] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time-Round-Robin: An O(1) Proportional Share Scheduler . In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC'11)*, 2011.

[73] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. of ATC*, 2013.

[74] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. of USENIX ATC*, 2013.

[75] S. Orathai and K. S. Hyong. Is Co-Scheduling Too Expensive for SMP VMs? In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*, 2011.

[76] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of ICDCS*, 1982.

[77] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the IEEE Distributed Compute System*, 1982.

[78] J. Ouyang and J. R. Lange. Preemptable ticket spinlocks: Improving consolidated performance. In *Proc. of JSSPP*, 1998.

[79] J. Ouyang and J. R. Lange. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment (VEE'13)*, 2013.

[80] S. Panneerselvam and M. M. Swift. Chameleon: Operating System Support for Dynamic Processors. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, 2012.

[81] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Service Networks: the Single-node Case. *ACM Transaction on Networking.*, 1, 1993.

[82] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Service Networks: the Multiple Node Case. *ACM Transaction on Networking.*, 2, 1994.

[83] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: A system for flexible parallel execution. In *Proc. of PLDI*, 2012.

[84] J. Rao, K. Wang, X. Zhou, and C. Xu. Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*, 2013.

[85] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *Proc. of HPCA*, 2013.

[86] J. Rao and X. Zhou. Towards fair and efficient smp virtual machine scheduling. In *Proc. of PPoPP*, 2014.

[87] J. Rao and X. Zhou. Towards Fair and Efficient SMP Virtual Machine Scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'14)*, 2014.

[88] J. Rao and X. Zhou. Towards fair and efficient smp virtual machine scheduling. In *Proc. of PPoPP*, 2014.

[89] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'95)*, 1995.

[90] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12))*, 2012.

[91] P. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic coscheduling on workstation clusters. In *Proc. of JSSPP*, 1998.

[92] X. Song, H. Chen, and B. Zang. Characterizing the performance and scalability of many-core applications on virtualized platforms. Technical Report FDUPPITR-2010-002, Parallel Processing Institute, Fudan University, 2010.

[93] X. Song, S. Jicheng, C. Haibo, and Z. Binyu. Scheduling processes, not vcpus. In *Proc. of APsys*, 2013.

[94] Spec Java Server Benchmark. http://www.spec.org/jbb2005.

[95] S. Sridharan, G. Gupta, and G. S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *Proc. of ICS*, 2013.

[96] S. Sridharan, G. Gupta, and G. S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proc. of PLDI*, 2014.

[97] I. Stoica, H. Adbel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of Multimedia Computing and Networking*, 1996.

[98] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for smp vms? In *Proc. of EuroSys*, 2011.

[99] O. Sukwong and S. H. Kim. Is co-scheduling too expensive for smp vms? In *Proc. of Eurosys*, 2011.

[100] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. In *Proc. of ASPLOS*, 2013.

[101] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. The Locker Holder and the Lock Waiter Pre-preemption Problems: Nip Them in the Bud Using Informed Spinlocks(I- Spinlock). In *Proceedings of 20th European Conference on Computer Systems*, 2017.

[102] The CPU Scheduler in VMware vSphere 5.1. http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.

[103] The Princeton Application Repository for Shared-Memory Computers (PARSEC) . http://parsec.cs.princeton.edu/.

[104] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of VM*, 2004.

[105] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of VM*, 2004.

[106] VMware. VMware Horizon View Architecture Planning 6.0. In *VMware Technical White Paper*, 2014.

[107] VMware. Vmware horizon view architecture planning 6.0. In *VMware Technical White Paper*, 2014.

[108] 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 (OSDI'94)*, 1994.

[109] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Determinstic Proportional-share Resource Management. In *MIT Technical Report*, 1995.

[110] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proc. of PACT*, 2006.

[111] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06))*, 2006.

[112] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proc. of HPDC*, 2011.

[113] 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 (HPDC'11)*, 2011.

[114] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proc. of HPDC*, 2011.

[115] C. Weng, Z. Zhang, M. Li, and X. Lu. The hybrid framework for virtual machine systems. In *Proc. of VEE*, 2009.

[116] C. Weng, Z. Zhang, 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 Environment (VEE'09)*, 2009.

[117] Windows Azure Open Cloud Platform. http://www.windowsazure.com.

[118] Xen Credit Scheduler. http://wiki.xen.org/wiki/Credit_Scheduler.

[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 *Proc. of HPDC*, 2012.

[120] 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 21th International Symposium on High Performance Parallel and Distributed Computing (HPDC'12))*, 2012.

[121] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proc. of ISCA*, 2013.

[122] H. Zhang and S. Keshav. Comparison of Rate-Based Service Dispilines. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'91)*, 1991.

[123] L. Zhang. VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'90)*, 1990.

[124] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proc. of EuroSys*, 2013.

[125] Y. Zhao, J. Rao, and Q. Yi. Characterizing and Optimizing the Performance of Multithreaded Programs Under Interference. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques (PACT'16)*, 2016.

[126] Y. Zhao, K. Suo, L. Cheng, and J. Rao. Scheduler Activations for Interference-resilient SMP Virtual Machine Scheduling . In *Proceedings of the 18th ACM/IFIP/USENIX Conference on Middleware (Middleware'17)*, 2017.

[127] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. *J. Comput. Secur.*, 2013.

BIOGRAPHICAL STATEMENT

Yong Zhao was born in Luo Yang on Jan 10th, 1986. He earned B.E. degree from East China Jiao Tong University in 2009 and an M.S. degree from Institute of Software, Chinese Academy of Sciences in 2013 respectively, and a Ph.D. degree from the Computer Science and Engineering Department at the University of Texas at Arlington in 2019. His research interests lies in operating systems, virtualization and computer architecture and his work has been published in prestigious conferences about the computer systems such as SIGMETRICS, PACT, Middleware, SoCC and INFOCOM.