Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

HIGH PERFORMANCE CLOUD COMPUTING ON MULTICORE COMPUTERS

by
Jianchen Shan

The cloud has become a major computing platform, with virtualization being a key to allow applications to run and share the resources in the cloud. A wide spectrum of applications need to process large amounts of data at high speeds in the cloud, e.g., analyzing customer data to find out purchase behavior, processing location data to determine geographical trends, or mining social media data to assess brand sentiment. To achieve high performance, these applications create and use multiple threads running on multicore processors. However, existing virtualization technology cannot support the efficient execution of such applications on virtual machines, making them suffer poor and unstable performance in the cloud.

Targeting multi-threaded applications, the dissertation analyzes and diagnoses their performance issues on virtual machines, and designs practical solutions to improve their performance. The dissertation makes the following contributions. First, the dissertation conducts extensive experiments with standard multicore applications, in order to evaluate the performance overhead on virtualization systems and diagnose the causing factors. Second, focusing on one main source of the performance overhead, excessive spinning, the dissertation designs and evaluates a holistic solution to make effective utilization of the hardware virtualization support in processors to reduce excessive spinning with low cost. Third, focusing on application scalability, which is the most important performance feature for multi-threaded applications, the dissertation models application scalability in virtual machines and analyzes how application scalability changes with virtualization and resource sharing. Based on the modeling and analysis, the dissertation identifies key application features and system

factors that have impacts on application scalability, and reveals possible approaches for improving scalability. Forth, the dissertation explores one approach to improving application scalability by making fully utilization of virtual resources of each virtual machine. The general idea is to match the workload distribution among the virtual CPUs in a virtual machine and the virtual CPU resource of the virtual machine manager.

HIGH PERFORMANCE CLOUD COMPUTING ON MULTICORE COMPUTERS

by
Jianchen Shan

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

Department of Computer Science

May 2018

APPROVAL PAGE

HIGH PERFORMANCE CLOUD COMPUTING ON MULTICORE COMPUTERS

Jianchen Shan

Xiaoning Ding, PhD, Dissertation Advisor — Date
Assistant Professor, Computer Science, New Jersey Institute of Technology

---

Cristian Borcea, PhD, Committee Member — Date
Professor, Computer Science, New Jersey Institute of Technology

---

Narain Gehani, PhD, Committee Member — Date
Professor, Computer Science, New Jersey Institute of Technology

---

Reza Curtmola, PhD, Committee Member — Date
Associate Professor, Computer Science, New Jersey Institute of Technology

---

Qing Liu, PhD, Committee Member — Date
Assistant Professor, Electrical and Computer Engineering, New Jersey Institute of Technology

BIOGRAPHICAL SKETCH

Author: Jianchen Shan

Degree: Doctor of Philosophy

Date: May 2018

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, USA, 2018
- Master of Science in Computer Science,
  Shanghai University, Shanghai, China, 2011
- Bachelor of Science in Computer Science,
  Shanghai University, Shanghai, China, 2008

Major: Computer Science

Presentations and Publications:

J. Shan, X. Ding, "Dynamically adjusting virtual CPU features to avoid low and unstable performance in big VMs". *(under submission)*

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

N. R. Paiker, J. Shan, C. Borcea, N. Gehani, R. Curtmola, X. Ding, "Design and implementation of an overlay file system for cloud-assisted mobile apps", *in IEEE Transactions on Cloud Computing (TCC)*, 2018.

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

X. Ding, J. Shan, S. Jiang, "A general approach to scalable buffer pool management", *in IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2016.

P. Neog, H. Debnath, J. Shan, N. Paiker, N. Gehani, R. Curtmola, X. Ding, C. Borcea, "FaceDate: A mobile cloud computing app for people matching", *in 7th International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (Mobilware 2016)*, 2016.

J. Shan, X. Ding, N. Gehani, "APPLES: Efficiently handling spin-lock synchronization on virtualized platforms", *in IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

J. Shan, N. R. Paiker, X. Ding, N. Gehani, R. Curtmola, C. Borcea, "An overlay file system for cloud-assisted mobile applications", *in 32nd International Conference on Massive Storage Systems and Technology (MSST 2016)*, 2016.

J. Shan, X. Ding, N. Gehani, "APLE: Addressing lock holder preemption problem with high efficiency", *in 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom 2015)*, 2015.

X. Ding, J. Shan, "Diagnosing virtualization overhead for multi-threaded computation on multicore platforms", *in 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom 2015)*, 2015.

X. Ding, P. B. Gibbons, M. A. Kozuch, J. Shan, "Gleaner: Mitigating the blocked-Waiter wakeup problem for virtualized multicore applications", *in 2014 USENIX Annual Technical Conference (USENIX ATC 2014)*, 2014.

J. Shan, Y. Lei, "A novel parallel algorithm for near-field computation in N-body problem on GPU", *in 2011 IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS 2011)*, 2011.

J. Shan, Y. Lei, J. Zhu, "The algorithm mapping of the near-field computation in N-body problem on GPU ", *in 2011 International Conference on Computers, Communications,Control and Automation (CCCA 2011)*, 2011.

J. Zhu, Y. Lei, J. Shan, "Parallel FMM algorithm based on space decomposition", *in 2010 9th International Conference on Grid and Cloud Computing (GCC 2010)*, 2010.

To my beloved parents

单正海，赵玉花

ACKNOWLEDGMENT

First, I would like to express my deep and sincere gratitude to my doctoral advisor, Dr. Xiaoning Ding. I have been very fortunate to become his first Ph.D student and to get the opportunity to work closely with him. Being my role model, his enthusiasm, hard working, and courage to face challenges strongly motivated me. Give a man a fish, and you feed him for a day; teach a man to fish, and you feed him for a lifetime. His mentorship not only leaded me to the world of research but also made me an independent researcher and thinker, which would benefit me for a lifetime. Being my advisor, he has been always there to help and to give advice, from identifying research topics, designing experiments, to revising papers and improving presentation skills. He always holds a high and strict standard on our research work while being patient with my learning process and giving me freedom to choose research interest. His understanding, wide knowledge, and wisdom have been invaluable to me. Moreover, I sincerely appreciate his warm care and kind support that helped me going through the hard time of my life in the united states. As an international student studying oversea without family nearby, I think him not only as my advisor but also as my family.

I also wish to thank my collaborators: Dr. Cristian Borcea, Dr. Narain Gehani, Dr. Reza Curtmola, Dr. Hillol Debnath, Nafize Paiker, Weiwei Jia and Pradyumna Neog. It has been my true pleasure to work together with them. The fruitful discussions with them have been of a great value for me to broaden my perspective on how brainstorming can be so effective. In particular, I thank Dr. Cristian Borcea, the chair of the Computer Science department. He gave me the opportunity to work in the excellent and stimulating academic environment in his group. I am very grateful that he has extended my research scope and introduced me to the mobile cloud computing. I thank Dr. Narain Gehani and Dr. Reza Curtmola who made

constructive suggestions and feedbacks to my research that improved my scientific thinking and technical writing. I thank Dr. Hillol Debnath, Nafize Paiker, Weiwei Jia and Pradyumna Neog for exchanging ideas and discussing technical problems with me. They are also my lab mates who gave me their kind supports and friendships. For me, they are not only lab mates but also members in a big family.

I would like to thank all committee members of my Ph.D. dissertation for their help and advice. Especially, I thank Dr. Qing Liu from the Department of Electrical and Computer Engineering at NJIT for serving as the external committee member. I would also like to thank many anonymous conference and journal reviewers who have given me constructive comments on improving the quality of my papers.

Finally, I owe a lot to my parents, Zhenghai Shan and Yuhua Zhao, for all the sacrifices they have made to provide me the best education and support. I could not imagine how I could have gone through the toughest moments in these years, if I had not got their unconditional and endless love.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

Cloud computing has become main stream with virtualization being a cornerstone technology. With virtualization, virtual machines (VMs) are created as an abstraction of physical resources and complete execution environments for applications; applications from different users can be encapsulated into different virtual machines and be consolidated on the same physical machine to reduce cost and improve efficiency.

In this multi-core era, with increasing core count and memory size, the computational capacity of a physical machine keeps growing. At the same time, the demand for computing power in the cloud keeps increasing. A wide spectrum of applications need to process large amounts of data at high speeds in the cloud, e.g., analyzing customer data to find out purchase behavior, processing location data to determine geographical trends, or mining social media data to assess brand sentiment. To satisfy the demand for increasing computing power in each VM and to utilize the growing computational capacity of underlying physical machines, virtual machine sizes also grow steadily. For example, Amazon EC2 platform now provides virtual machines with 128 virtual CPUs and 3904 GiB memory.

To achieve high performance on virtual machines, similar to the executions on physical machines, applications usually create multiple threads and distribute computation to these threads. These threads run on multiple virtual CPUs in the VM, which are in turn scheduled on multiple cores in the physical machine, in order to utilize the computing power of multiple cores. Most VMs in the cloud are now with multiple virtual CPUs, and this percentage still keeps increasing. It is imperative to

study how multi-threaded applications perform in such VMs and ensure that high performance can really be achieved in the cloud.

Despite the similarity in the architectures (e.g., a VM with multiple virtual CPUs vs. a physical machine with multiple physical CPUs/cores) and execution environments (e.g., OSs and libraries), the execution and the performance of a multi-threaded application on a VM can be substantially different from those on a physical machine. It has been noticed that multi-threaded applications suffer serious and unpredictable performance degradation on VMs. The causes of such difference and performance issues include an additional layer of software (i.e., overhead introduced by virtualization) and the resource sharing and resource contention between the VMs hosted on the same physical machine. While these factors also affect the executions of single-thread applications, performance impact is particularly significant for multi-threaded applications, because of the special execution features of multi-threaded applications (e.g., synchronization and communications during their executions).

The performance issues with multi-threaded applications on VMs are also due to the lack of support at different system layers to fully consider the execution features of these applications. For example, at the hardware layer, many efforts have been paid to implement various hardware assistance to eliminate virtualization overhead for single-thread executions. However, the virtualization overhead caused by multi-threaded executions, such as the virtualization overhead associated with synchronization and communication, has not received enough attention, and there is little hardware assistance to effectively reduce such virtualization overhead.

At the virtual machine manager layer, virtual CPUs are not built to provision CPU resource to applications in an efficient way. For example, a virtual CPU is now implemented as an entity that is independently scheduled on a physical core and time-share the core with other virtual CPUs on the core. Thus, the virtual CPU cannot run continuously, and its computational capacity varies over time. This makes

it being unable to meet the expectation of multi-threaded applications for threads making continuous and steady progress. Also, virtual CPUs in a virtual machine are built to have symmetric performance. But such symmetry doesn't necessarily fit the application's resource demand. All these factors can significant impact the execution of multi-threaded applications on virtual machines.

Inside virtual machines, task scheduling at the guest OS layer is designed for physical cores, and is not aware of the non-continuity and varying computing capacity of the virtual CPUs. It cannot distribute workloads to virtual CPUs based on their activity and capability, and thus cannot fully utilize the allocated CPU resource to achieve high performance.

Targeting multi-threaded applications, the dissertation analyzes and diagnoses their performance issues on virtual machines, and designs practical solutions to improve their performance. First, focusing on the execution features of multi-threaded applications, the dissertation conducts extensive experiments with standard multicore applications, in order to evaluate the performance overhead on virtualization systems and diagnose the causing factors. The dissertation identifies the hardware assistance required to eliminate the virtualization overhead for multi-threaded applications to achieve high performance on virtual machines. Second, one of main sources of the performance overhead for multi-threaded applications is excessive spinning caused by spin-based synchronization, such as spin-locks. Focusing on excessive spinning, the dissertation designs and evaluates a holistic solution to make effective utilization of the hardware virtualization support in processors to reduce excessive spinning with low cost. Third, focusing on application scalability, which is the most important performance feature for multi-threaded applications, the dissertation models application scalability in virtual machines and analyzes how application scalability changes with virtualization and resource sharing. Based on the modeling and analysis, the dissertation identifies key application features and system factors

that have impacts on application scalability, and reveals possible approaches for improving scalability. Forth, the dissertation explores one approach to improving application scalability by making fully utilization of virtual resources of each virtual machine. The general idea is to match the workload distribution among the virtual CPUs in a virtual machine and the virtual CPU resource of the virtual machine manager.

The research contributions are summarized in the following sections.

1.2 Contributions of Dissertation

1.2.1 Diagnosing the Virtualization Overhead in the Multicore VMs

Hardware-assisted virtualization, as an effective approach to low virtualization overhead, has been dominantly used. However, existing hardware assistance mainly focuses on single-thread performance. Much less attention has been paid to facilitate the efficient interaction between threads, which is critical to the execution of multi-threaded computation on virtualized multicore platforms. We aim to answer two questions: 1) what is the performance impact of virtualization on multi-threaded computation, and 2) what are the factors impeding multi-threaded computation from gaining full speed on virtualized platforms. Targeting the first question, we measure the virtualization overhead for computation-intensive applications that are designed for multicore processors. We show that some multicore applications still suffer significant performance losses in virtual machines. Even with hardware assistance for reducing virtualization overhead fully enabled, the execution time may be increased by more than 150% when the system is not over-committed, and the system throughput can be reduced by 6x when the system is over-committed. To answer the second question, with experiments, we diagnose the main causes for the performance losses. Focusing the interaction between threads and between VCPUs, we identify and examine a few performance factors, including the intervention of the

virtual machine monitor (VMM) to schedule/switch virtual CPUs (VCPUs) and to handle interrupts required by inter-core communication, excessive spinning in user space, and cache-unaware data sharing.

1.2.2 Reducing the Synchronization Overhead in Multicore VMs

Spin-locks are widely used in software for efficient synchronization. However, they cause serious performance degradation on virtualized platforms, such as the Lock Holder Preemption (LHP) problem and the Lock Waiter Preemption (LWP) problem, due to excessive spinning by virtual CPUs (VCPUs). The excessive spinning occurs when a VCPU waits to acquire a spin-lock. To address the performance degradation, hardware facilities, such as Intel PLE and AMD PF, are provided on processors to preempt VCPUs when they spin excessively. Although these facilities have been predominantly used on mainstream virtualization systems, using them in a manner that achieves the highest performance is still a challenging issue. There are two core problems in using these hardware facilities to reduce excessive spinning. One is to determine the best time to preempt a spinning VCPU (i.e., the selection of spinning thresholds). The other is which VCPU should be scheduled to run after the spinning VCPU is descheduled. Due to the semantic gap between different software layers, the virtual machine monitor (VMM) does not have information about the computation characteristics on VCPUs, which is needed to address the above problems. This makes the problems inherently challenging. We propose a framework named AdPtive Pause-Loop Exiting and Scheduling (APPLES) to address these problems. APPLES monitors the overhead caused by excessive spinning and preempting spinning VCPUs, and periodically adjusts spinning thresholds to reduce the overhead. APPLES also evaluates and schedules "ready" VCPUs in a VM by their potential to reduce the spinning incurred by the spin-lock synchronization. The evaluation is based on the causality and the time of VCPU preemptions. The implementation of APPLES

incurs only minimal changes to existing systems (about 100 lines of code in KVM). Experiments show that APPLES can improve performance by 3% - 49% (14% on average) for the workloads with frequent spin-lock operations.

1.2.3 Analyzing the Application Scalability in the Multicore VMs

Virtual machine (VM) sizes keep increasing in the cloud. However, little attention has been paid to analyze and understand the scalability of multicore applications on big VMs with multiple virtual CPUs (VCPUs), assuming that application scalability on VMs can be analyzed in the same ways as that on physical machines (PMs). We demonstrate that, since hardware CPU resource is dynamically allocated to VCPUs, the executions of multicore applications on VMs show different scalability from those on PMs. We systematically study how the virtualization of CPU resource changes execution scalability, identifies key application features and system factors that affect execution scalability on VMs, and investigates possible directions to improve scalability.

We present a few important findings. First, the execution scalability of applications on VMs is determined by different factors than those on PMs. Second, virtualization and resource sharing can improve scalability by nature. Thus, applications may show better scalability on VMs than on PMs. Linear scalability can be achieved even when there is substantial sequential computation. Third, there is still much space to further improve execution scalability by enhancing system designs. Better scalability can be achieved by increasing allocation period length and/or matching resource allocation and workload distribution.

1.2.4 Improving the Application Scalability in the Multicore VMs

Hosting big virtual machines (VMs) with multiple virtual CPUs (VCPUs) on multicore servers has become a norm in modern cloud computing infrastructures.

However, multicore programs may suffer significant performance degradation and unpredictable performance variation on such VMs, when the VMs are co-located with other VMs. One of the major causes is the CPU utilization problem of the computation on such VMs, i.e., low CPU utilization causing performance degradation and unpredictable CPU utilization causing performance variation. The CPU utilization problem is in turn caused by the mismatch between the allocation of CPU time to VCPUs and the demand for CPU time of the computation distributed on the VCPUs.

We proposes dynamically adjusting the performance features of the VCPUs in each VM, including the amount of CPU time distributed to the VCPUs and the responsiveness of the VCPUs, based on the CPU time demand of the computation on each VCPU. The objective is to match the allocation of CPU time to VCPUs with the demand for CPU time, such that the computation in the VM can proceed at the highest speed possible, which is usually determined by the fair share of CPU time made available to the VM. We provides a system solution to adjust VCPU performance features, evaluates it with extensive experiments using PARSEC and TPC-W-like workloads, and demonstrates that it can effectively avoid performance degradation and performance variation of multicore programs on virtualized platforms, and can also improve overall system throughput.

1.3 Structure of Dissertation

The rest of the dissertation is organized as follows. Chapter 1.3 analyzes and identifies the virtualization overhead for the multicore applications in the cloud. Chapter 2.6 focus on the synchronization overhead in cloud. It introduces the APPLES framework to efficiently handle the spin-lock synchronization and reduce the excessive spinning. Chapter 3.5 reveals how scalability is affected by the virtualization system in the cloud and the directions to improve it. Chapter 4.6 describes how dynamic asymmetric

virtual CPUs matches the resource allocation and workload distribution for desired scalability in the cloud. We conclude this dissertation in Chapter 6.

CHAPTER 2

DIAGNOSING VIRTUALIZATION OVERHEAD FOR MULTI-THREADED COMPUTATION ON MULTICORE PLATFORMS

2.1 Introduction

Applications usually have lower performance on virtual machines than on physical machines, due to the overhead introduced by virtualization. Virtualization overhead is one of the major concerns when people consolidate their workloads using virtual machines (VMs) or migrate their workloads into virtualized clouds. Processors, as primary system resources, are usually first evaluated before other resources. Thus, identifying and reducing CPU virtualization overhead are a main focus of virtualization technology [1, 2, 3, 4, 5, 6].

Hardware-assisted virtualization is an effective method to reduce virtualization overhead, and has been widely used in almost all mainstream virtualization platforms. Hardware assistance, especially that from hardware processors (e.g. Intel VT-x [7] and AMD-V [3]), makes virtual devices behave and perform identically to the corresponding hardware devices for improved performance. However, existing hardware assistance for CPU virtualization is mainly focused on single thread performance. While various types of hardware support has been developed to accelerate each individual thread (e.g., the support for nonfaulting accesses to privileged states and the support for accelerating address translation), little attention has been paid to efficient multi-threaded execution on virtual machines, especially the efficient interaction between threads. CPU virtualization usually incurs minimal performance penalty for single-thread applications on latest processors. But, as this chapter will show, multi-threaded applications may suffer substantial performance losses, even with the hardware assistance for reducing virtualization overhead fully enabled.

For example, due to the lack of facilitates to efficiently coordinate VCPUs, a multicore processor is usually virtualized into a set of single-core virtual CPUs (VCPUs) that are scheduled independently by the virtual machine monitor (VMM). This mismatch between multicore processors and virtual CPUs may not slow down single-thread applications. But it penalizes multi-threaded applications, which are designed and optimized for multicore processors and expect VCPUs to behave identically to real computing cores.

This chapter measures and diagnoses the execution overhead of multi-threaded applications on virtualized multicore platforms with the latest hardware assistance for virtualization. With the maturity of hardware-assisted virtualization, virtualization overhead has been significantly reduced for single-thread executions, and the intent of further reducing the virtualization overhead for computation-intensive applications is losing its momentum recently. With the measurement, we want to motivate architects and system designers to further reduce virtualization overhead for multi-threaded applications, and with the diagnosis, we want to find out a few promising directions for developing new techniques and/or optimizing existing designs. The contributions of this chapter are as follows.

First, this chapter shows that, while single-thread computation has decent performance on virtual machines, multi-threaded computation still suffer significant performance losses. The execution time may be increased by more than 150%, even when the host system is not over-committed. The performance loss is not due to resource sharing or contention. When the host system is over-committed, the overhead increases significantly and the system throughput may be reduced by as much as 6x. This clearly shows that there is still strong demand for further reducing the virtualization overhead for computation-intensive applications.

Then, with experiments, this chapter reveals a few factors degrading the performance of multi-threaded computation on virtualized multicore platforms. As

far as we know, some factors have not been identified or studied in other literatures. Specifically, this chapter identifies the following performance-degrading factors: 1) VCPU rescheduling/switching overhead incurred by VCPU state changes; 2) the overhead incurred by handling inter-processor interrupts (IPIs) cannot be eliminated even with hardware support such as Advanced Programmable Interrupt Controller virtualization (APICv) [8]; 3) excessive VCPU spinning in user space cannot be eliminated with hardware support such as Pause-Loop Exiting (PLE) [9]; 4) VCPU rescheduling/switching overhead incurred by preempting spinning VCPUs; 5) opaque cache architectures in virtual machines prevent efficient data sharing among threads.

Finally, this chapter discusses a few techniques that can be used to reduce the overhead caused by the above factors. To our best knowledge, this is the first work that systematically measures the virtualization overhead and diagnoses the performance degradation of multi-threaded applications on the systems with the latest hardware assistance for efficient virtualization.

2.2 Experimental Settings and Methodology

We conducted our experiments on two Dell PowerEdge servers. One is a R720 server with 64GB of DRAM and two 2.40GHz Intel Xeon E5-2665 processors, each of which has 8 cores. The other is a R420 server with 48GB of DRAM and a 2.50GHz Intel Xeon E5-2430 V2 processor with 6 Ivy Bridge-EN cores. We created virtual machines on the servers. The VMM is KVM [10]. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. CPU power management can reduce application performance on VMs [11]. To prevent such performance degradation, in the experiments, we disabled the C states other than C0 and C1 of the processors, which have long switching latencies.

We selected the benchmarks in PARSEC 3.0 and SPLASH2X suites in the PARSEC benchmark package [12][1]. We compiled the PARSEC and SPLASH2X

benchmarks using gcc with the default settings of the *gcc-pthreads* configuration in PARSEC 3.0. We used the *parsecmgmt* tool in the PARSEC package to run them with native inputs. In the experiments, unless stated otherwise, When we ran a benchmark in a VM we set the minimum number of threads in each benchmark equal to the number of VCPUs in the VM with the "-n" option. We pre-warmed the buffer cache in the guest operating system to minimize I/O operations. Please note that the memory capacity (16GB) of a VM is large enough to buffer the input and output data sets of the benchmark and to provide the memory space for its execution.

We carried out two groups of experiments. In the first group of experiments, we ran benchmarks with default system settings. The hardware assistance for reducing virtualization overhead (e.g. Extended Page Tables (EPT) and Pause-Loop Exiting (PLE)) was fully enabled in KVM. We ran each benchmark under three different scenarios: 1) on a VM with dedicated hardware resources, 2) on multiple VMs sharing hardware resources and with one instance of the benchmark running in each VM, and 3) on the physical machine hosting the VMs. With these experiments, we want to compare the performance of the benchmark under these scenarios and show the overhead incurred by virtualization.

In the second group of experiments, we reran the benchmarks suffering large performance degradation. We want to diagnose the root causes for the performance degradation and reveal the factors causing virtualization overhead. In the experiments, we used the following methods to diagnose the executions. In some experiments, we temporarily changed some system settings when we execute a benchmark. We selected the settings that can remove or alleviate certain types of virtualization overhead. For example, by disabling PLE support, we can reduce the overhead due to handling the VM_EXITs triggered by PLE events. In some other experiments, we used the *perf* tool for Linux and KVM to profile the executions [13].

[1]We did not select benchmark *cholesky* in SPLASH2X since its execution time is too short (less than 0.01s) and varies significantly across different runs.

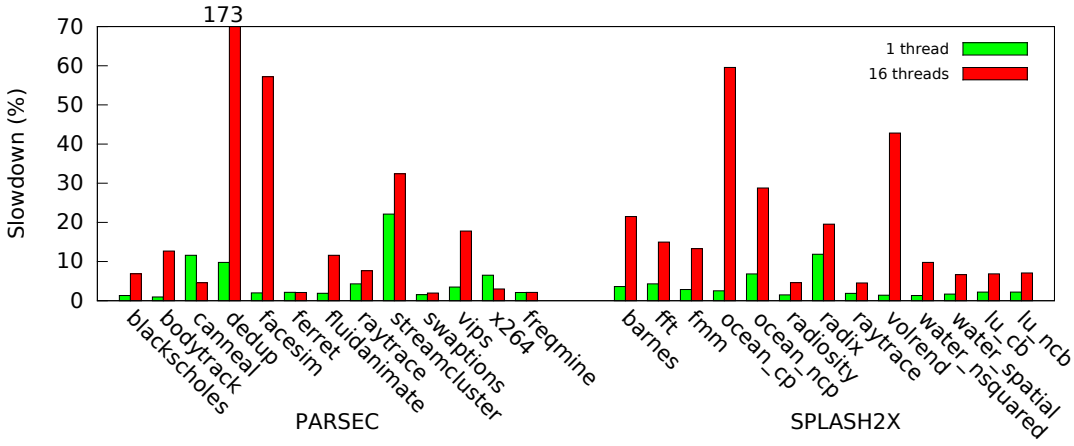

Figure 2.1 Slowdowns of PARSEC benchmarks and SPLASH2X benchmarks in a 16-VCPU virtual machine relative to their executions on the 16-core R720 server.

In some cases, neither of the above methods could identify the root causes. In these cases, we tried to manually modify the benchmarks and examine the performance difference.

2.3 Measuring Virtualization Overhead

This section shows the virtualization overhead of the benchmarks under two different scenarios. First, we measure the virtualization overhead when the physical machine is not over-subscribed. Only one VM was launched in the experiments, and the number of VCPUs was equal to the number of cores on the physical machine hosting the VM. We compare the performance of the benchmarks on the VM against that on the physical machine.

Figure 2.1 shows the slowdowns of the benchmarks due to virtualization on the R720 server for both single-thread executions (i.e., -n 1) and multi-threaded executions (16 threads are used, i.e., -n 16). The figure clearly shows that multi-threaded executions were slowed down on virtual machines by much larger percentages than single-threaded executions. On average, these benchmarks were slowed down by 4% with single-thread executions and by 21% with 16-thread

executions. The slowdowns of multi-threaded executions vary across the benchmarks in a very large range, from less than 1% (*canneal*, *radiosity*, and *lu_ncb*) to more than 150% (*dedup*). While half of the benchmarks were slowed down slightly by less than 10%, seven benchmarks were slowed down substantially by more than than 20%, and three benchmarks were slowed down by more than 50%.

Then, we measure the virtualization overhead when the physical machine is over-subscribed. We launched multiple VMs and run an instance of the benchmark in each VM. We set the number of VCPUs in each VM equal to the number of cores in the physical machine and set the number of threads in each instance equal to the number of the VCPUs in a VM. Since we launched multiple VMs, the physical cores were time-shared by VMs. Thus, instead of using the performance of each individual benchmark instance, we use system throughput to analyze virtualization overhead. Specifically, we use *Weighted-Speedup* to measure the system throughput, which is the aggregated speedup of the benchmark instances. The speedup is relative to the execution of the benchmark on a VM when the system is not over-subscribed. Thus, the scenario with only one VM launched and one instance running on the VM serves as the baseline, and the throughput under the baseline scenario is 1. For example, if there are two instances of the benchmark running on two VMs and the execution time of the benchmark is doubled, the Weighted-Speedup is also 1 (i.e., 0.5+0.5), indicating that the throughput is the same as that under the baseline scenario. A weighted-speedup larger than 1 indicates higher throughput than the baseline.

In the experiments, we gradually increased the number of VMs (and the number of benchmark instances) from 1 to 4 before the physical memory is filled. Figure 2.2 shows how the system throughput changes for the benchmarks which suffer high virtualization overhead (slowed down by more than 20% when the system is not over-subscribed). Since VCPUs might not be always active when these benchmarks ran, the system was not fully loaded when there were fewer active VCPUs than

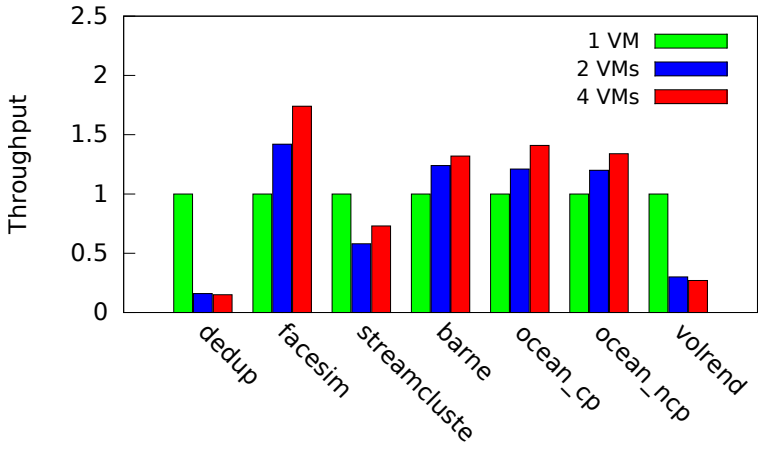

Figure 2.2 Throughput of PARSEC benchmarks and SPLASH2X benchmarks when the number of VMs was increased from 1 to 4

physical cores. Increasing the number of VMs helped making a full utilization of the hardware resources and thus led to higher throughput. We observed this trend with some benchmarks. For example, the system throughput was increased by 74% for *facesim* when the number of VMs was increased to 4.

However, we also observed that, with a few benchmarks, the system throughput reduced dramatically when there was more than one VM. For example, when the number of VMs was increased to 2, surprisingly the throughputs of *dedup*, *streamcluster*, and *volrend*, were reduced by about 6x, 2x, and 3x, respectively. Please note that, since the baseline is the performance with the system hosting 1 VM, the performance degradation is in addition to that incurred by the virtualization overhead in the baseline scenario.

Under both scenarios, the performance degradation was measured when the same amount of physical resource was used (i.e., all the resource on the physical machine). Thus, the performance degradation was due to virtualization overhead, instead of short of physical resource. The experiments evidently show that the virtualization overhead is still high for some multicore applications and must be effectively reduced. While some execution overhead is expected on virtual

machines, the large performance degradation observed in the above experiments is not normal and makes virtualized platforms an inefficient choice for some multi-threaded workloads.

To better understand the virtualization overhead, we have investigated the possible causes for the performance degradation. Since I/O operations are minimized and memory resources are not oversubscribed, we concentrate on examining the factors related to the virtualization of hardware resources on processors. Because only some multi-threaded executions show large slowdowns, we do not investigate the factors that affect both single-thread and multi-threaded applications (e.g., increased pressure on TLBs due to the adoption of techniques such as EPT). Instead, we focus on the factors related to the interaction between threads and between VCPUs.

2.4 Diagnosing Virtualization Overhead

In this section, we analyze and diagnose the performance degradation of the multi-threaded applications running on virtual machines. We want to find out the factors degrading performance and to what degree they can degrade performance. Thus, we select the workloads with large performance degradation in the experiments in the previous section.

We focus our investigation on the interaction between threads and between VCPUs. Specifically, threads may interact with each other using various types of IPCs. They may also share or exchange data through shared memory space. Processors/cores usually rely on Inter Processor Interrupts (IPIs) to coordinate with each other. They access shared data in shared caches. If there are multiple caches holding multiple copies of shared data, they must keep the copies consistent. With experiments, we reveal that IPCs, IPIs, and data sharing can incur high virtualization overhead in different ways on virtual machines. In the following several subsections, we first isolate the factors degrading performance and examine their overhead when

the system is not over-subscribed. Then we analyze the executions with the system over-subscribed with multiple VMs.

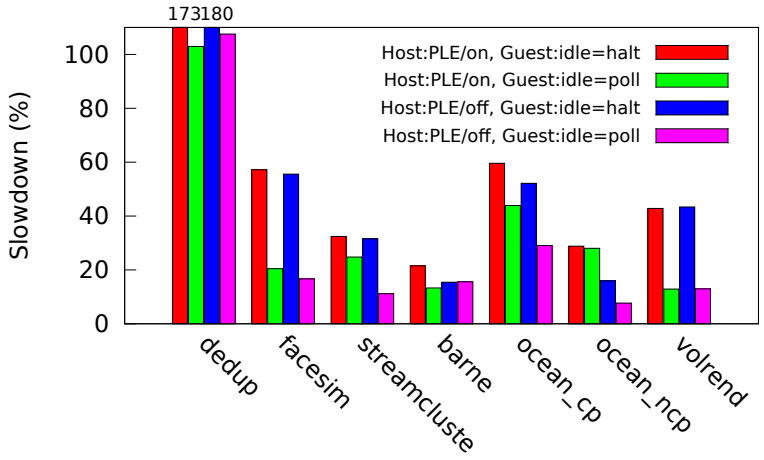

Figure 2.3 Slowdowns of the benchmarks are reduced after the overhead incurred by switching/rescheduling idle VCPUs and spinning VCPUs is removed.

2.4.1 Overhead Due to Switching/Rescheduling Idle VCPUs

Multi-threaded computation usually runs on multiple VCPUs in a virtual machine. Some VCPUs become idle when there lacks runnable tasks, and are activated when some tasks become runnable. To make efficient utilization of hardware resources, the VMM must be notified to handle these state changes of VCPUs. The overhead is thus incurred.

Frequent VCPU state changes can be caused by *blocking* synchronization, with which a thread waiting for an event blocks itself by giving up its execution resources (mainly the CPU) spontaneously. A blocked thread relies on the operating system to wake it up when the event happens. Blocking makes the number of *active* threads in a virtual machine change dynamically. The number of VCPUs employed by these threads also changes accordingly. When the number of active threads drops below the number of active VCPUs, some VCPUs will become idle. When the number of active threads increases beyond the number of active VCPUs, idle VCPU must be

activated. For example, when a thread calls *pthread_mutex_lock()* to request a mutex that is held by another thread, it will block itself through appropriate library/system calls, waiting for the release of the mutex. If there are no other threads ready to run in the system, the VCPU running the thread becomes idle. In the guest OS, an idle VCPU executes the idle loop, which typically calls a special instruction (e.g., `HLT` on Intel 64 and IA-32 architecture ("x86") platforms). When the mutex is released, the threads waiting for it are woken up. To maximize throughput, the guest OS may activate idle VCPUs to schedule waking threads onto them.

In a virtualized environment, the special instruction and the operations to activate idle VCPUs must be handled by the VMM, even though they would be carried out directly by hardware in a non-virtualized environment. When software issues the special instruction to place a particular VCPU into the idle state, the core running the VCPU will raise an exception and trap into the VMM. The VMM may take this opportunity to reschedule other VCPUs onto this idling core. When a thread is ready to run on an idle VCPU, the VMM must activate the VCPU and reschedule it onto a physical core. These operations incur much higher cost (e.g., usually a few microseconds) than those required in a non-virtualized environment to switching an idle core back (e.g., switching from C1 to C0 states takes no more than 1 microsecond on contemporary Intel Xeon CPUs).

To evaluate the overhead caused by switching and rescheduling idle VCPUs, we change the idling operation in the guest OS. Instead of having an idle VCPU call `HLT` instruction, we make it enter a polling idle loop. In this way, the overhead incurred by descheduling and rescheduling idle VCPUs can be avoided. Thus, the overhead can be indicated by comparing the performance of the benchmarks before and after the change.

We select the benchmarks with slowdowns larger than 20% with the default idling operation, and re-run them with polling idle loop. Figure 2.3 compares

the slowdowns of the benchmarks with different idling operations. By removing the overhead of descheduling and rescheduling idle VCPUs (polling idle loop), the slowdowns of the benchmarks can be significantly reduced[2]. The average slowdown is reduced from 59% to 35%. Among these benchmarks, *dedup* receives the largest performance improvement, and its slowdown is reduced from 173% to 103%. The slowdown of *volrend* is reduced by the largest percentage (about 2/3 of the slowdown is removed).

In real practice, the performance degradation due to handling idle VCPUs can be reduced by reducing the cost of context switches. There have been some enhancements adopted in KVM to reduce such cost (e.g., by reducing the cost of saving and restoring FPUs) [14]. For this reason, compared to the measurement that we performed earlier [15], handling idle VCPUs now causes smaller performance degradation. This shows the effectiveness of these enhancements. However, the experiments in this section also show that the overhead of handling idle VCPUs can still cause significant performance degradation to some applications and should be further reduced.

2.4.2 Overhead Due to Switching/Rescheduling Spinning VCPUs

After the overhead to handle idle VCPUs has been removed, the benchmarks still suffer some performance losses. To identify the causes, we continued to examine the overhead caused by switching and rescheduling spinning VCPUs.

VCPU spinning is usually caused by *spinning* synchronization, with which a thread repeatedly checks some condition (e.g., the value of a shared variable) to determine if it can continue. The spinning may be initiated explicitly by the program, and the thread remains in user space during spinning. It may also be initiated by

[2]Note that system setting changes in this section are for diagnosis purposes and cannot be applied to general practice. While some changes may be used to improve performance in some specific scenarios (e.g., when the system is under-subscribed), they may cause serious performance degradation in other scenarios (e.g., when the system is over-subscribed).

the OS kernel when the execution of the thread traps into the kernel. On virtual machines, spinning may cause the Lock-Holder Preemption problem (LHP). LHP happens when a VCPU is descheduled from the host platform while it is holding a lock. Since the VCPU is descheduled, it cannot proceed and the lock cannot be released quickly. Thus, other VCPUs that are waiting on the lock must spin until this descheduled VCPU is rescheduled. The spinning, however, prevents the descheduled VCPU from being rescheduled quickly. This forms a situation of live-lock and significantly reduces system throughput. This live-lock situation may also be caused by spinning in synchronization primitives other than spinlocks (e.g., barriers) on virtualized platforms. For brevity, we use LHP-like problems to refer to the lock holder preemption problem and other similar problems caused by spinning[3].

To deal with LHP-like problems, hardware solutions (such as Intel pause-loop-exiting (PLE) support [16]) have been implemented on processors. They detect VCPUs that have been spinning for a while and preempt these VCPUs. Thus, the VMM can involve to reallocate the resources to other VCPUs that can make progress, e.g., the VCPUs holding the locks.

However, spinning is usually used to replace blocking in synchronization primitives for higher performance. Preempting spinning VCPUs actually changes spinning back to blocking. Since the hardware support, such as PLE, preempts VCPUs only based on the lengths of spinning periods, it may degrade performance if spinning VCPUs are preempted when there are not LHP-like problems. For example, when CPU cores are not over-subscribed, LHP-like problems will not happen. Even on an over-subscribed system, it is still possible that spinning VCPUs are preempted

[3]Synchronization primitives may combine spinning and blocking operations — a thread spins for a period of time, and if the expected event has not happen, it blocks itself. Usually, the spinning lasts only a brief period of time. Thus, the spinning will not cause LHP-like problems, and the hardware support (e.g. PLE) dealing with LHP-like problems does not detect or interrupt such short-period spinning. Since only blocking operations incur virtualization overhead with this combined approach, we does not consider the spinning in these synchronization primitives.

when they are about to finish spinning. In such cases, preempting spinning VCPUs introduces unnecessary overhead.

To test whether the PLE support causes any performance degradation, we disabled PLE support in KVM and re-ran the experiments. The slowdowns of the benchmarks (relative to their executions on the physical machine) are shown in Figure 2.3. When PLE support is disabled, the performance is only slightly improved. When polling idle loop is used, the average slowdown is lowered to 29% (from 35%) by disabling PLE support. With the default idling operation, disabling PLE reduces the average slowdown to 56% (from 59%). Disabling PLE support is most effective for *ocean_ncp*, which only suffers the virtualization overhead caused by preempting spinning VCPUs. By disabling PLE, its slowdown can be reduced from 28% to 8%.

The experiments show that preempting spinning VCPUs can slightly reduce performance in the cases where there are no LHP-like problems. For a small number of applications such as *ocean_ncp*, it may substantially degrade performance. When the number of VCPUs in a VM keeps increasing in the future (e.g., Amazon EC2 now provides instances with 40 VCPUs), synchronization will become more frequent and lock contention will also be more intensive. This may increase the chances of spinning VCPUs being preempted, as well as the performance degradation. For example, people have observed that it takes 369s to boot a 80-VCPU VM with PLE enabled, while it takes only 25s with PLE disabled [17].

2.4.3 Overhead Due to Inter-VCPU Coordination

We notice that, after removing the virtualization overhead caused by handling idle VCPUs and spinning VCPUs, though the slowdowns are substantially lowered (from 59% to 29% on average), the selected benchmarks still suffer some performance degradation on virtual machines. The average slowdown is higher than that of their single-thread executions (7%). This is largely due to *dedup*, which suffers a

108% slowdown with 16-thread executions but only 10% slowdown with single-thread executions. For the selected benchmarks other than *dedup*, though their 16-thread executions are also slowed down by larger percentages than their single-thread executions, the differences between the slowdowns are not as significant as *dedup*. Without *dedup*, the average slowdown is 15% for 16-thread executions and 6% for single-thread executions.

To identify the factors causing the remaining slowdowns, especially that of *dedup*, we used *perf* to profile the executions of the benchmarks, and found that most VM_EXITs were caused by the accesses to Advanced Programmable Interrupt Controller (APIC). These APIC accesses are mainly incurred by sending and receiving rescheduling inter-processor interrupts (IPIs) and TLB shutdown IPIs. A rescheduling IPI is for a CPU to notify another CPU to perform rescheduling. This usually happens when there is a thread to be activated on the recipient CPU. A TLB shutdown IPI is for a CPU to notify other CPUs to flush TLB entries (i.e., "TLB shutdown" This usually happens when a CPU flushing a TLB entry needs to flush the TLB entries on other CPUs. When a CPU receives an IPI, it must acknowledge (ACK). Then, it signals End-Of-Interrupt (EOI) at the completion of the interrupt service. On a physical machine, the OS sends and receives IPIs, as well as the ACKs and EOIs, by accessing APIC registers, and the APIC hardware delivers them. But on virtual machines, the VMM must intercept the accesses, process the requests, and deliver the IPIs/ACKs/EOIs. This makes the operations much more expensive on virtual machines than on physical machines.

With existing hardware design and system software design, the overhead caused by APIC accesses cannot be completely isolated. To estimate the overhead, we leverage the APIC virtualization (APICv) support introduced recently in Ivy Bridge-EP processors [8]. The support reduces the overhead of hardware interrupts on virtual machines by processing some operations relating interrupts and APIC (e.g.,

read accesses) in hardware without triggering VM_EXITs. Since the APICv support is not available on the R720 server, we repeat the experiments on the R420 server. To clearly demonstrate the overhead of APIC accesses, the PLE support is turned off in KVM, and the polling idle loop is selected in the VM. With the experiments, we compare the performance of the benchmarks with APICv turned off and on.

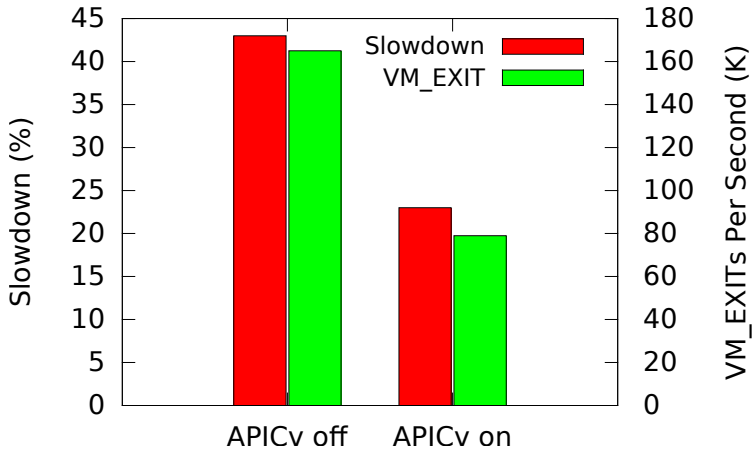

Figure 2.4 Slowdowns of *dedup* and the numbers of VM_EXITs per second incurred by APIC accesses when APICv is turned off and on. The number of VCPUs in the VM and the number of threads in *dedup* are 4.

We are most interested in the performance of *dedup*, since it has the largest slowdown and can show the overhead incurred by APIC accesses more clearly than other benchmarks. Figure 2.4 shows the slowdowns of *dedup* and the numbers of VM_EXITs per second due to APIC accesses. With APICv enabled, the number of VM_EXITs is reduced by 52%. The slowdown of *dedup* is reduced roughly proportionally by 47%. However, even with APICv enabled, *dedup* still incurs frequent VM_EXITs (about 20K VM_EXITs per second on each core) due to frequent APIC accesses, and thus still suffers substantial performance degradation. The experiments show that the VM_EXITs caused by APIC accesses can significantly reduce application performance on virtual machines. Though APICv can help reducing the cost, there is still much space for further improvement.

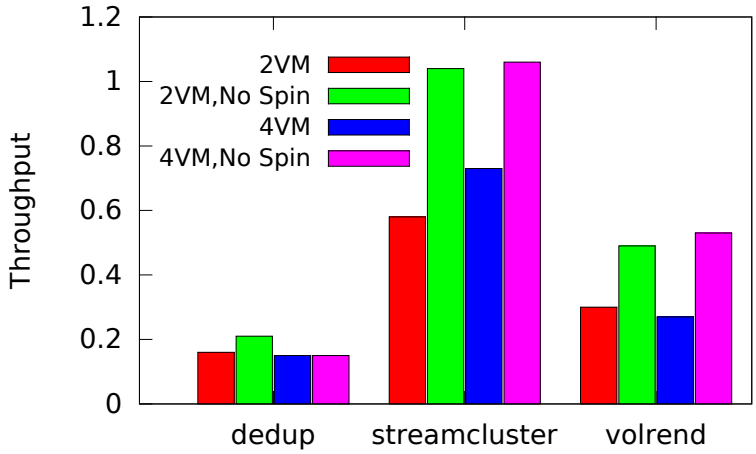

Figure 2.5 Throughput of *dedup*, *streamcluster*, and *volrend* when the system is oversubscribed.

2.4.4 Overhead Due to Spinning in User Space

In this subsection, we investigate the throughput degradation when the system is over-subscribed with multiple VMs running *dedup*, *streamcluster*, or *volrend*. We were surprised to observe their dramatic performance degradation shown in Figure 2.2. By carefully profiling the execution of these benchmarks, we found a significant portion of execution time was spent on spinning with *streamcluster* and *volrend*, though the PLE support was enabled. It turned out that PLE only detects and preempts VCPUs spinning in kernel mode (CPL=0) [9]. For spinning synchronization in user space (e.g., *pthread_spin_lock*), PLE cannot help preventing LHP-like problems.

To isolate the performance degradation due to the spinning in user space, we manually modified the source code of these three benchmarks and replaced spinning synchronization with blocking synchronization. As shown in Figure 2.5, the throughput of *streamcluster* is increased to 1.04 with 2 VMs and 1.06 with 4VMs, indicating that the performance degradation with the stock *streamcluster* benchmark is mainly caused by VCPU spinning in user space. Though the throughput of *volrend* is increased by almost 2x, it is still much lower than 1. This indicates that VCPU spinning at the user level is one of the major factors for the throughput

degradation. For *dedup*, VCPU spinning is not the major cause for the degradation. its throughput is only increased by 30% after the modification. Profiling shows that *dedup* spends over 85% of its execution time inside the guest OS kernel calling function *smp_call_function_many*, which sends IPIs to VCPUs to do operations such as TLB shootdowns. The main cause of the throughput degradation of *dedup* is that the system is overwhelmed by processing APIC accesses and routing IPIs.

2.4.5 Overhead due to Cache-Unaware Virtualization

Existing virtualization technology gives little consideration or support to cache optimization and management on virtual machines. For example, the actual architecture of hardware caches is not available on virtual machines. The information about cache resources available to a VCPU is either opaque or misleading. Although this simplifies the design of VMMs, it complicates cache optimization in virtual machines or makes it impossible to do cache optimization. For example, cache-aware scheduling in Linux [18] and cache-aware task group [19] need concrete knowledge on cache structure. With existing virtualization technology, these techniques can hardly be performed on virtual machines.

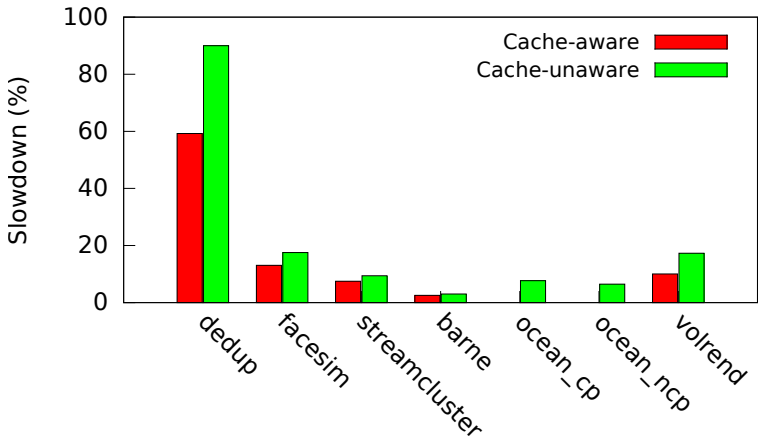

Figure 2.6 Comparison of the slowdowns of the benchmarks when threads in the same benchmark instance share the last level cache and when they do not.

To illustrate the performance loss caused by cache-unaware virtualization, we perform the following experiments. We launch two instances of the same benchmark and run them in parallel. The minimum number of threads in each instance is set to 8. We run the instances in three scenarios: (1) on the 16-core R720 server with one instance on each processor; (2) on a 16-VCPU virtual machine with the threads in the same instance scheduled on the VCPUs running on the cores of the same physical processor, and (3) on a 16-VCPU virtual machine without any restriction on the VMM scheduling VCPUs or the guest OS scheduling threads. In scenario 2, the threads in the same instance can share the last level cache (LLC) on the processor, while in scenario 3 they may not.

For each benchmark, we calculate the slowdowns of its executions in scenario 2 and scenario 3, relative to its execution in scenario 1. Figure 2.6 compares the slowdowns. In scenario 2, by sharing the last level cache, the threads in the same benchmark instance can exchange data more efficiently and incur less traffic between the two CPU sockets. Thus, the executions show higher performance in scenario 2. Among these benchmarks, *dedup*'s slowdown is reduced by the largest percentage. This is because *dedup* uses a pipelined programming model and most of its data is shared by the threads working at different pipeline stages. In scenario 3, without cache sharing information, the threads in the same benchmark instance cannot be scheduled to the VCPUs sharing the LLC. They cannot exchange and share data efficiently. Thus, the executions have larger slowdowns in scenario 3.

2.5 Summary and Discussion

The experiments show that multi-threaded computation still suffers significant performance degradation on SMP VMs. Even when the system is not over-subscribed, the execution of a multi-threaded application can be slowed down by over 150%. When the system is over-subscribed, the throughput can be reduced by as much as 6x.

Recently, reducing the virtualization overhead of I/O operations attracts increasingly more attention, and reducing the virtualization overhead of CPU resources is losing its momentum. The measurement in this chapter show that CPU virtualization can incur larger overhead than I/O virtualization (about 35% for I/O-intensive workloads [20]). Thus, reducing the virtualization overhead of CPU resources should be paid more attention, especially for multi-threaded applications.

Reducing CPU virtualization overhead is important not only because there are workloads suffering dramatic performance loss, but also because an increasing number of applications will be multi-threaded and computation-intensive. With the growing density and dropping prices of DRAM, it becomes cost-effective to build commodity servers with hundreds of gigabytes even terabytes of DRAM. With such memory capacities, the data sets of most applications can be completely saved or mostly buffered in memory. New memory types, e.g, phase-change memory, will be non-volatile and have even higher densities than DRAM. In the future, memory may save all the data sets and become the "new disk" for a large proportion of workloads. This trend is reflected by the rapid advancement of in-memory computing technology. With minimal I/O operations, the performance of these workloads will be largely determined by how they utilize multicore processors to process their data in memory. Minimizing virtualization overhead for multi-threaded computation is critical for their performance in the cloud.

With experiments, we show that, though single-thread executions have decent performance on virtual machines, the interaction between threads incurs large overhead, which dramatically degrades multi-threaded executions on virtual machines. On one hand, due to the lack of appropriate hardware support, the interaction between threads involves the intervention from the VMM. Specifically, the VMM needs to handle the state changes of VCPUs and other events (e.g., IPIs) incurred by inter-thread interaction and synchronization, while the corresponding

events on physical machines are handled by hardware. On the other hand, the behavioral differences between VCPUs and real CPUs make conventional optimization for efficient interaction and synchronization between threads (e.g., spinlocks, data sharing through shared caches) ineffective on virtual machines. Existing virtualization technology lacks effective methods to address these problems. For example, even though PLE is used to address the LHP problem, it may incur some performance degradation in some cases, and cannot be used to stop excessive spinning in user space.

The performance degradation of multi-threaded computation on virtual machines would be more serious if care is not taken. With the core count on each CPU socket keeping increasing, applications must split their work and distribute tasks among more threads to get performance improvement. However, this may incur more frequent synchronization to coordinate the tasks and more interaction between the tasks, which in turn cause higher performance degradation to the executions in the cloud.

Though software approaches (e.g., smarter VCPU scheduling algorithms) may alleviate the performance degradation, fundamentally addressing the problems (e.g., that with APIC accesses) is beyond the capability of software approaches. The most effective approach would be the enhancements in hardware CPU designs. While there are a few factors degrading the performance of multi-threaded executions, the root cause is that software must explicitly coordinate CPU resource sharing (e.g., deschedule idle/spinning VCPU, routing IPIs to idle VCPUs, etc). Thus, a fundamental solution would be using hardware to coordinate the resource sharing among VCPUs. For example, a physical core can be designed to have multiple "logical cores", one for each VCPU, and share the hardware resources on the physical core among these logical cores. Similar ideas have been used in I/O virtualization (e.g., SR-IOV allows an I/O device to function as multiple separate physical devices). The

idea is also used in SMT processors to hide memory latencies. But different with SMT design, which allows hardware threads to share CPU resources in a fine time granularity at the instruction level, the "logical cores" for virtualization can share CPU resources at coarse granularities (e.g., microseconds) to simplify the design and improve scalability.

2.6 Related Work

A number of early studies have identified the performance issues associated with VMM intervention and management complexity, such as privilege instructions and memory address translation [1, 7]. Most of these issues have been addressed or effectively alleviated with the enhancements in hardware designs.

Regarding CPU virtualization, most recent studies focus on the overhead caused by the lock holder preemption (LHP) and other similar problems [4, 21, 5, 6, 22, 16, 23, 24]. On current platforms, approaches with hardware assistance (e.g. Intel PLE [16] and AMD PF [25]) to detect and preempt spinning VCPUs have become *de facto* standard solutions. This chapter does not focus on LHP or LHP-like problems. Instead, it studies the virtualization overhead incurred by the solutions and the performance losses due to the limitation of the solutions.

The virtualization overhead caused by blocking synchronization is identified and analyzed in [15, 26, 27]. This chapter quantifies the overhead in more situations and with the newer software system that has integrated a few enhancements for reducing the overhead [14]. Besides the overhead caused by blocking synchronization, this chapter also quantifies the overhead caused by other factors.

In this chapter, we show that the opacity of hardware cache architecture on VCPUs leads to slow memory accesses and degrades performance. Regarding memory accesses in virtual machines, research has been conducted to reduce the overhead of address translation [2, 28]. The non-uniformity of memory latencies was found to

affect the performance of virtualized systems [29]. Memory space overhead is another consideration in memory virtualization. For example, ballooning and deduplication techniques have been developed to reduce the space overhead [30, 31].

Virtualization overhead is a major consideration for people choosing virtualized platforms. There are studies to measure and identify virtualization overhead for different workloads, e.g., HPC workloads [32, 33, 34] and databases [28], to compare the performance of different virtualization infrastructures [35, 36], or to compare virtualized and non-virtualized infrastructures [37]. We focuses on the overhead caused by CPU virtualization for multi-threaded computation-intensive workloads.

Some studies focus on the overhead incurred by I/O operations [20, 29, 38, 39]. They are remotely related with the work.

CHAPTER 3

APPLES: EFFICIENTLY HANDLING SPIN-LOCK SYNCHRONIZATION ON VIRTUALIZED PLATFORMS

3.1 Background and Motivation

In this section, we first introduce the problems caused by spin-locks in virtual machines. Then, we introduce the hardware facilities in processors for dealing with these problems, and explain how existing virtualization systems utilize these facilities, using Kernel-based Virtual Machine (KVM) and Intel PLE support as examples. We show that these hardware facilities must be better utilized by VMMs to achieve higher performance[1].

3.1.1 Problems Caused by Spin-locks in VMs

Spin-locks are usually used to protect short critical sections. While waiting to acquire a spin-lock, a thread repeatedly checks the availability of the lock, because the waiting is expected to be brief. With spinning, a lock can be acquired as soon as it is released. At the same time, because the thread does not block itself, the costly overhead associated with context switches is avoided.

Ticket spin-lock is a special type of spin-lock that guarantees the order of lock acquisitions to provide fairness and avoid starvation among lock requests. A ticket spin-lock uses a queue to manage the requests for the lock and schedules the requests accordingly. Thus, a lock waiter cannot acquire the lock until the lock waiter before it on the queue releases the lock.

In a virtualized environment, because of the scheduling of VCPUs, a thread running on a VCPU may not be able to continuously make progress as it does on

[1]Although the description in this section and the APPLES design later in the chapter are mainly based on Intel PLE, they can be applied directly or with slight modification to the systems with AMD PF or other similar hardware utilities, which detect and stop spinning based on the thresholds set by the VMM.

a PCPU. When a VCPU is preempted, the thread running on it also stops. Thus, if a thread is holding a spin-lock and the VCPU is preempted, the spin-lock cannot be released quickly until the VCPU is rescheduled. Thus, other threads waiting for the lock have to spin for unexpected long time. This is the lock holder preemption (LHP) problem. The spinning causes a live-lock situation, where spinning VCPUs hold CPU resources and wait for the lock, and the lock holder VCPU waits for CPU resources to resume execution. If the spinning cannot be stopped promptly, system throughput may be significantly reduced.

With ticket spin-lock, the situation is more complex. The live-lock situation may be caused by not only lock-holders but also lock-waiters. When the VCPU of a lock waiter is preempted, all the subsequent lock waiters on the queue have to spin for unexpected long time until the lock waiter is rescheduled, even though the lock itself may be released during the spinning. This is defined as the lock waiter preemption (LWP) problem.

3.1.2 Hardware Facilities in Processors to Control Excessive VCPU Spinning

Modern processors provide hardware support for virtualization to reduce overhead. On these processors, PLE and other similar facilities are designed to control excessive VCPU spinning. With PLE, a processor first detects spinning VCPUs by examining the instructions executed by the VCPUs. On X86 architecture, spin-lock primitives usually repeatedly call PAUSE instructions to implement spinning. To detect spinning, the processor checks the intervals (in number of CPU cycles) between consecutive PAUSE instructions executed by a VCPU. For a spinning VCPU, the intervals are very short, since the VCPU only checks the condition for stopping spinning between PAUSE instructions. Thus, the processor compares the lengths of the intervals against a pre-set parameter *PLE_gap*. If the lengths do not exceed

PLE_gap, it determines that the VCPU is spinning. If no PAUSE instruction is executed in an interval of *PLE_gap*, it determines that the spinning stops.

When the spinning is continuing, the processor needs to determine whether the spinning should be stopped. For this purpose, it keeps track of the length of the spinning by counting the number of cycles spent on PAUSE instructions and the intervals between PAUSE instructions. If the length of the spinning exceeds a pre-set spinning threshold *PLE_window*, the processor will trigger a VM_EXIT to stop the spinning and transfer the control to the VMM, so that the VMM can deschedule the spinning VCPU and reschedule another VCPU.

AMD Pause Filter (PF) functions similar to the Intel PLE. It also checks intervals between consecutive PAUSE instructions and considers PAUSE instructions with intervals smaller than *PAUSE Filter Threshold* to be in the same loop. It interrupts and reports to the OS the spin loops exceeding *PAUSE Filter Count* intervals. Both *PAUSE Filter Threshold* and *PAUSE Filter Count* are pre-set by software. For AMD PF, *PAUSE Filter Count* acts as the spinning threshold. Because of the similarity, we pick one — Intel PLE for our APPLES design and experiments. But APPLES applies equally well to the systems with AMD PF.

3.1.3 The Utilization of the Hardware Facilities in VMM

With PLE, when the two parameters *PLE_gap* and *PLE_window* are set, the processor detects and interrupts spinning VCPUs autonomously. The VMM controls the PLE facility by adjusting these parameters. It is relatively easy to find an adequate value for *PLE_gap* since PAUSE instructions are called much more frequently in spin-locks than in other scenarios. For example, KVM sets *PLE_gap* to 128 cycles by default, which proves to be effective in practice. Thus, we does not discuss the adjustment of *PLE_gap* parameter, and focuses only on how to find an adequate value for the spinning threshold *PLE_window*.

Besides adjusting the parameters, the VMM must also handle VM_EXITs caused by PLE facilities. The VMM takes the chances to preempt spinning VCPUs, put them onto the "ready" VCPU list, and schedule other VCPUs. For example, when a spinning VCPU of a VM is preempted, KVM examines the "ready" VCPUs in the same VM. If it can find a "ready" VCPU, which was not preempted due to spinning, KVM schedules the VCPU. For brevity, this case is called "successful yielding", since the spinning CPU "yields" the processor to a VCPU that can make progress. Otherwise, KVM reschedules the VCPU that is just preempted. This case is called "unsuccessful yielding".

The adjustment of spinning thresholds and the selection of VCPUs to schedule in are two key problems that a VMM must solve to make effective utilization of the hardware facilities controlling VCPU spinning. As we will show later using KVM as an example, these two problems are challenging, and many ad-hoc methods have been tested in existing VMMs. However, workloads with frequent spin-lock synchronization still suffer substantial performance degradation on virtualized platforms.

Methods to Adjust Spinning Thresholds in KVM In the past, KVM would use a system-wide spinning threshold and set it to a fixed value selected empirically based on the normal spinning time under some typical workloads. The spinning time is measured when the VCPUs running the workloads are not preempted. Thus, a threshold can be set slightly higher than the normal spinning time, and any spinning longer than this threshold is considered as abnormal, indicating the occurrence of LHP or LWP problems.

A problem with a fixed spinning threshold was noticed. When physical CPUs (PCPUs) are under-subscribed, preempting spinning VCPUs cannot improve the utilization of PCPUs, and thus incurs unnecessary overhead. The overhead can be very high with large VMs. For example, experiments have shown that it takes

369s to boot a 80-VCPU VM with PLE enabled, while it takes only 25s with PLE disabled [17].

To improve the performance when physical CPUs (PCPUs) are not over-subscribed, attempts have been made to dynamically adjust spinning thresholds. The objectives are to increase the spinning threshold when PCPUs are under-subscribed and to restore the threshold when PCPUs are over-subscribed. For example, one of such attempts increases the threshold on "unsuccessful yieldings" and decreases it on "successful yieldings" [40]. The rationale is that "successful yieldings" indicate that there have been some non-spinning VCPUs preempted (i.e., PCPUs are over-subscribed) and "unsuccessful yieldings" indicate that there is not a "ready" VCPU waiting to be scheduled (i.e., PCPUs are under-subscribed).

In the latest design, KVM maintains a spinning threshold for each VCPU. If a VCPU is preempted and switched out because the VCPU runs out of the time slice, KVM determines that the VCPU is sharing a PCPU with other VCPUs, and quickly reduces the threshold of the VCPU to improve the utilization of the PCPU. This is to deal with the situation in which the PCPU is over-subscribed. For the situation in which the PCPU is under-subscribed, increasing the spinning threshold helps improving performance because this reduces the interruption to VCPU execution. Thus, when a VCPU is preempted because it spins and reaches the spinning threshold, KVM gradually increases its spinning threshold [41].

The above methods improve the performance when PCPUs are under-subscribed. But CPU over-subscription is a common practice [42]. These methods cannot appropriately adjust spinning thresholds when PCPUs are over-subscribed. We provide a quantitative illustration of the above problem using a few representative experiments. We select two benchmarks, *ebizzy* and *dbench*, and run them on a 16-core machine. (Please refer to Section 3.3 for benchmark description and machine configuration.) We use two 16-VCPU VMs. For each benchmark, we run two

instances of the benchmark in parallel on the two VMs, one on each VM, and collect the performance reported by the benchmark (throughput for *dbench* and execution time for *ebizzy*). We first run the benchmarks using the default KVM setting with the mechanism adjusting spinning thresholds enabled. We use this configuration as baseline. Then, we disable the mechanism. We use the same spinning threshold for the VCPUs in the two VMs and vary the spinning threshold from 512 cycles to 32768 cycles. We repeat the experiments for each of the different threshold values, and show the normalized performance relative to that with the baseline configuration in Figure 3.1.

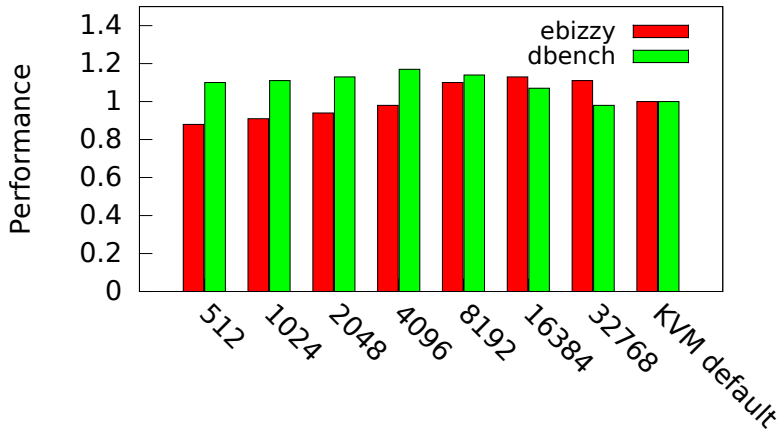

Figure 3.1 Normalized performance of *ebizzy* and *dbench* when the spinning threshold is varied from 512 cycles to 32768 cycles, relative to the performance with the *default* KVM configuration.

The figure clearly shows that the performance of the two benchmarks changes with the threshold. The performance of benchmark *dbench* varies from 0.98 to 1.17, and the performance of *ebizzy* varies from 0.88 to 1.13. At the same time, different benchmarks achieve the best performance with different spinning thresholds (4096 cycles for *dbench* and 16384 cycles for *ebizzy*). The experiments show that, to achieve optimal performance, spinning thresholds must be carefully tuned based on workloads.

However, the current KVM system cannot adjust the thresholds adequately, leading to suboptimal performance.

Candidate VCPU Selection in KVM When a VCPU is preempted because the spinning threshold is reached, the VMM must select a VCPU and schedule it on the vacated computing core. In KVM, a directed yield approach is used [43]. All the VCPUs in the same VM form a circle. The KVM searches the VCPUs other than the VCPU that is just preempted, following the circle. During the search, only "ready" VCPUs are considered, which include two types of VCPUs — the VCPUs preempted due to the depletion of time slices and the VCPUs preempted due to excessive spinning. For brevity, we call the first type of VCPUs *resource-waiter* VCPUs and the second type of VCPUs *lock-waiter* VCPUs[2].

Resource-waiter VCPUs are more likely to make progress than the lock-waiter VCPUs after rescheduled (i.e., granted with resources). In KVM, resource-waiter VCPUs are more preferred than lock-waiter VCPUs. When a resource-waiter VCPU is found, it is selected to run unconditionally. But, when a lock-waiter VCPU is found, it is selected to run if it is labeled as "checked"; otherwise, it is labeled as "checked" to be selected next time. The "checked" label is removed when the VCPU is scheduled.

When a VCPU is selected, its location in the circle is marked. Later, when more spinning VCPUs are preempted, new searches will start from this location. Thus, consecutive searches will traverse the circle and schedule resource-waiter VCPUs in the first round. Then, in the second round, lock-waiter VCPUs are also considered,

[2]It is possible that a VCPU depletes its time slice while it is spinning and waiting for a spin-lock. In such a case, the VCPU is "misclassified" as a resource-waiter VCPU. However, the possibility of such "misclassification" is slim, because spinning is capped by the spinning threshold and is very brief (usually shorter than 10 microseconds), and a time slice is much longer (at least a few milliseconds). At the same time, with existing hardware support, the VMM is not aware of VCPU spinning until it reaches the spinning threshold. Thus, it is not able to put spinning VCPUs into the lock-waiter category if they are preempted before they reach the spinning threshold.

because the mechanism assumes that the preempted lock holders have been scheduled in the first round and the lock-waiter VCPUs can continue to make progress when scheduled. If there are not VCPUs ready to run, KVM reschedules the VCPU that is just preempted (i.e., "unsuccessful yielding").

The main problem of the method is with the quality of the candidate VCPUs selected by the method. First, it checks the VCPUs in a VM based on the order in which they are organized in the circle, instead of the possibility of the VCPUs being the causes of excessive spinning. Excessive spinning is usually caused by waiting for preempted VCPUs, which are either holding spin-locks or waiting in ticket spin-lock queues before other VCPUs. These preempted VCPUs should be rescheduled before the VCPUs waiting for them and other VCPUs making new requests for the same spin-lock, so as to avoid additional spinning. Thus, quickly rescheduling these VCPUs is the most effective method to prevent excessive spinning.

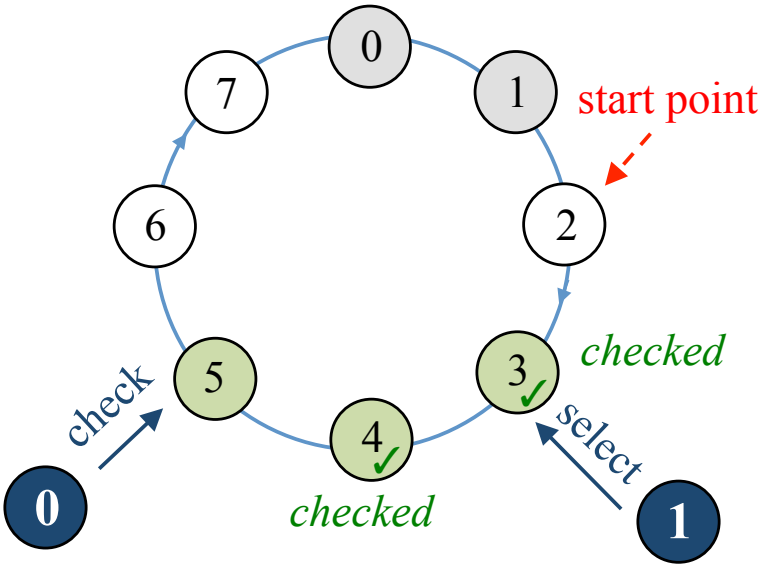

Figure 3.2 Candidate VCPU selection in KVM.

The current method in KVM cannot select VCPUs that are more likely to reduce excessive spinning. Even worse, though the method gives a slightly higher priority to resource-waiter VCPUs, there is still a high probability that lock-waiter VCPUs are

selected and they continue spinning after being rescheduled. This further decreases the quality of the candidate VCPUs selected. This problem is caused because there may be concurrent searches from the same location on the VCPU circle of a VM — a search starting earlier labels lock-waiter VCPUs as "checked" and another search starting later selects a "checked" VCPUs as a candidate VCPU before the earlier search finds and reschedules a resource-waiter VCPU.

This problem is as shown in Figure 3.2. In a VM with 8 VCPUs, VCPU #0 and VCPU #1 run on two different PCPUs. VCPU #0 is preempted and then VCPU #1 is preempted before a VCPU is selected to replace VCPU #0. Thus, both the PCPUs (i.e., the ones running and then preempting these two VCPUs) start searching from the same location (marked as "start point" in the figure). The PCPU preempting VCPU #0 starts earlier than the PCPU preempting VCPU #1. It checks VCPU #3 and VCPU #4, which are lock-waiter VCPUs, and labels them as "checked". Thus, the PCPU preempting VCPU #1 can select VCPU #3 as candidate VCPU and schedule it. With the low quality of VCPU candidates, potential VCPU spinning cannot be effectively reduced.

3.2 APPLES Design and Implementation

As we have introduced earlier, to utilize the spinning suppression hardware facilities equipped on current processors, there are two problems that must be solved: 1) the VMM must carefully adjust spinning thresholds for VMs; and 2) when the hardware facilities preempt a spinning VCPU, the VMM must select an appropriate VCPU to occupy the vacated PCPU. To address these problems, APPLES uses two components: APLE (Adaptive Pause-Loop Exiting) to dynamically adjust spinning threshold; and HVS (Heuristic VCPU Selection) to select candidate VCPUs when spinning VCPUs are preempted.

In this section, we introduce each component by first analyzing the problems and challenges and then describing its design. After that, we introduce the implementation of APPLES based on KVM and Linux.

3.2.1 APLE for Adjusting Spinning Thresholds

The adjustment of spinning threshold must make a difficult trade-off between different types costs and benefits, which makes it a challenging problem. On one hand, setting high thresholds increases excessive spinning and leads to low resource utilization. On the other hand, setting low thresholds may interrupt normal spinning prematurely. Spin-locks are used to protect short critical sections. Spinning ensures that a lock can be acquired as soon as it is released. At the same time, since spinning is expected to be brief, it incurs lower overhead than blocking, which is considered to be expensive because of the high cost of the context switches associated with blocking operations. Interrupting normal spinning increases synchronization overhead since it actually turns spin-based synchronizations into block-based synchronizations. If spinning thresholds are set too low, the VCPUs that are spinning normally may be preempted prematurely just before the lock holder is about to release the lock, incurring costly context switches between VCPUs. This can significantly increase synchronization overhead and reduce system throughput.

Possible Approaches and APLE Basic Idea When setting spinning thresholds, the VMM struggles between two conflicting objectives. One is to stop VCPU spinning as early as possible in case spinning VCPUs are waiting for other VCPUs temporarily preempted. The other is to avoid stopping VCPU spinning too early for efficient synchronization in case suspended VCPUs are not blocking spinning VCPUs from making progress.

An intuitive approach for adjusting spinning thresholds is to first determine the amount of time that a VCPU usually spends on spinning when the lock holding

VCPU is not preempted and then set the thresholds slightly higher than this amount. However, due to the semantic gap between system layers, it would be "mission impossible" to estimate the amount online. There are several reasons. Firstly, the VMM does not have information about lock operations in virtual machines. Thus, it is not possible for the VMM to *predict* the amount of spinning time (e.g., by profiling and modeling the execution of the workloads). Secondly, the VMM is not aware of VCPU spinning until it is notified when a processor stops the spinning exceeding the threshold. Thus, it is not possible for the VMM to determine adequate thresholds by *measuring* the amount of spinning[3]. Finally, when a processor stops a spinning VCPU, though the VMM knows the amount of spinning, it cannot determine whether the VCPU is waiting for a preempted VCPU or not. Thus, it still cannot *estimate* the amount of spinning when the LHP or LWP problem does not happen.

APLE is based on the following observations. If spinning thresholds are set too low, some overhead is caused because the time spent on spinning is wasted and extra time is used on descheduling spinning VCPUs and rescheduling other VCPUs. The overhead decreases if the thresholds are increased. If spinning thresholds are set too high, spinning VCPUs are preempted late. Overhead is caused by excessive spinning and descheduling and rescheduling VCPUs. The overhead decreases if smaller thresholds are used. Thus, optimal thresholds can be approached by varying the thresholds and choosing those leading to lower overhead.

APLE assumes that each workload runs in a VM and assigns a spinning threshold to each VM. It does not use a system-wide spinning threshold for all the VMs on the same physical machine, because different workloads have different locking behaviors and different spinning time before getting a lock. A threshold that achieves optimal performance for some workloads may cause serious performance degradation

[3]The spinning time may be measured with the collaboration from guest OSs [44], which is not available on public cloud.

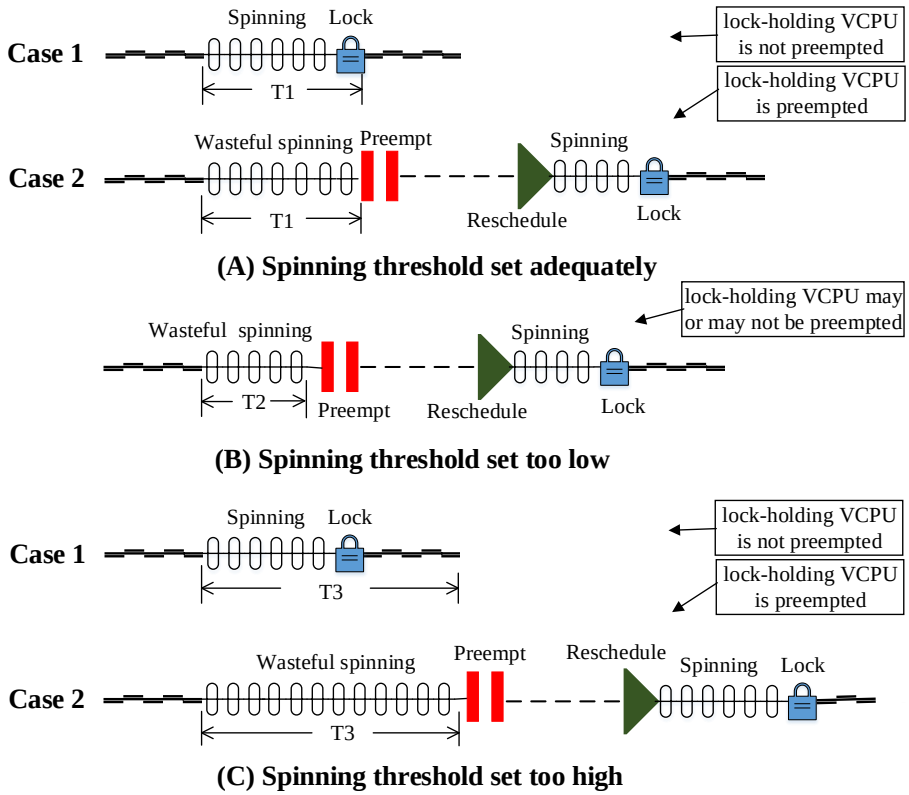

Figure 3.3 Overhead from wasteful spinning and wasteful VCPU switches under three scenarios, using the LHP problem as an example. The figure only shows the VCPU requesting a spin-lock. The lock-holding VCPU is not shown in the figure, but its status is shown in the boxes. A "pause" symbol (parallel vertical bars) indicates that the corresponding VCPU is preempted.

for other workloads. It does not use a per-VCPU spinning threshold because VCPUs sharing the same lock have similar locking behavior, e.g., every VCPU spins for longer time for longer critical section to finish. APLE also dynamically adjusts spinning thresholds to respond to workload changes in VMs.

Wasteful Spinning and Wasteful VCPU Switches We use the LHP problem as an example to explain the rationale behind APLE. In Figure 3.3, we compare the executions of a VCPU under three different scenarios: (a) when the spinning threshold is adequately set (Figure 3.3(A); (b) when the spinning threshold is set too low (Figure 3.3(B)); and (c) when the spinning threshold is set too high (Figure 3.3(C)). In the middle of the execution, the VCPU requests a spin-lock that is currently held

by another VCPU (not shown in the figure). Thus, it spins before it enters the critical section. However, the spinning incurs different overhead depending on the spinning threshold and whether the lock-holding VCPU has been preempted or not.

As illustrated in Figure 3.3(A), with the spinning threshold adequately set ($T1$), if the lock holding VCPU is not preempted, the spinning will not be interrupted before the lock is acquired. The spinning is considered *normal spinning*. In this case, the execution is exactly the same as that on a physical machine, and there is no overhead incurred. However, if the lock holding VCPU is preempted, the spinning will be stopped when it reaches the threshold, and the spinning VCPU is preempted. When the VCPU is rescheduled later, it still needs to spin and wait for the release of the lock. Since the spinning before the VCPU is preempted does not lead to a lock acquisition, it is considered *wasteful spinning*. Compared to the execution on a physical machine, the execution on the virtual machine incurs additional overhead due to the VCPU switch (i.e., descheduling the spinning VCPU and rescheduling another VCPU). Thus, the VCPU switch is a *wasteful VCPU switch*.

As illustrated in Figure 3.3(B), if the spinning threshold is set too low ($T2$), the VCPU may be stopped prematurely, even when the lock holding VCPU is not preempted. This incurs the overhead through wasteful spinning and wasteful VCPU switches. Compared to the scenario shown in Figure 3.3(A) (the spinning threshold is adequately set), setting the threshold too low increases the chance that the spinning VCPU is preempted. If the spinning VCPU is preempted, next time when it is scheduled, the lock may still not be available, though the lock-holder may have changed. Thus, the VCPU must start over to wait for the release of the lock. With a low threshold, it may be preempted prematurely again. It is possible that the VCPU is descheduled and rescheduled multiple times before it gets the lock, incurring more wasteful spinning and VCPU switches.

If the spinning threshold is set too high ($T3$), as shown in Figure 3.3(C), the execution is similar to that in scenario (A), when the lock-holding VCPU is not preempted. But, if the lock-holding VCPU is preempted in the case when the spinning threshold is set higher than that in scenario (A), the VCPU spins for longer time before its is preempted. Compared to scenario (A), the spinning incurs higher overhead from wasteful spinning.

Among these three scenarios, no matter whether the threshold is set too low or too high, higher overhead will be caused, compared to an adequately set threshold. Therefore, the overhead can be a reliable indicator of the level of the threshold.

Both wasteful spinning and wasteful VCPU switches are visible to and handled by the VMM. Thus, their overhead can be accurately measured in the VMM with low cost. This is one of the advantages of APLE. Specifically, the overhead of wasteful spinning can be determined by spinning thresholds and the number of times the thresholds reached. The overhead of each VCPU switch is the time between the corresponding VM_EXIT and VM_ENTRY events.

The Calculation of Inefficiency as a Metric To adjust the threshold, APLE measures the overhead caused by wasteful spinning and wasteful VCPU switches for each VM. However, the amount of overhead cannot be directly used in the adjustment, because the overhead is affected by the factors other than the spinning threshold. For example, the resources allocated to a VM change over time on a over-committed system. With more resources (e.g., more PCPUs) allocated to a VM, the workload on it makes faster progress and incurs higher overhead at the same time.

APLE calculates *inefficiency*, which is the ratio between the time spent on wasteful spinning and wasteful VCPU switches and the PCPU time consumed by the VCPUs. APLE calculates inefficiency periodically and uses it as the metric for the adjustment. Each time period is called an *epoch*. In each epoch, APLE collects the

CPU time allocated to the VM. It also maintains a counter counting PLE events, which it resets at the beginning of each epoch. Each time spinning reaches the threshold, in the VM_EXIT event handler (for PLE events), APLE increments the counter, and timestamps the beginning and end of PLE event handling. At the end of each epoch, APLE calculates the overhead of wasteful spinning by multiplying the spinning threshold with the value in the counter, and calculates the overhead of VCPU switches by adding the time spent by PLE event handling. Then, it divides the sum of the two types of overhead by the total CPU time allocated to the VM, the result being the inefficiency of the VM in the epoch.

APLE Algorithm To achieve the best performance, with APLE, the VMM periodically measures the inefficiency, and adjusts the spinning threshold to minimize the inefficiency using the APLE Algorithm below.

When a VM is launched, this algorithm sets an initial value of the desired threshold T_d (e.g., 8192 in our experiments). While running, the VM tries the desired threshold and the thresholds slightly lower and slightly higher than the desired threshold, once for an epoch. For fast adjustment, the difference between these thresholds δ cannot be too small. However, to keep the threshold close to the optimal value, δ cannot be too large either. Based on our experiments, a value between 512 and 2014 works best for the adjustment. At the end of each epoch, APLE calculates the inefficiency of the epoch. When these epochs with different thresholds finish, APLE compares the inefficiency of these epochs. It uses the threshold of the epoch with the smallest inefficiency to update the the desired threshold. Then, the desired threshold is used for the next round of adjustment.

Epoch lengths vary dynamically based on the frequency at which VCPUs are preempted due to excessive spinning (i.e., the frequency of VM_EXITs incurred by PLE events on Intel platforms). Specifically, each epoch corresponds to a fixed

Algorithm 1 APLE Algorithm

T_d: desired spinning threshold of a VM

T_0: initial spinning threshold of the VM

T_u: upper bound for the spinning threshold of the VM

T_l: lower bound for the spinning threshold of the VM

$T_d \leftarrow T_0$

while the VM is running **do**

set the spinning threshold of the VM to T_d

wait for the finish of an epoch E_1, and calculate the inefficiency of the VM in E_1

set the spinning threshold of the VM to $min(T_u, T_d + \delta)$

wait for the finish of an epoch E_2, and calculate the inefficiency of the VM in E_2

set the spinning threshold of the VM to $max(T_l, T_d - \delta)$

wait for the finish of an epoch E_3, and calculate the inefficiency of the VM in E_3

compare the inefficiency of epochs E_1, E_2, and E_3

$T_d \leftarrow$ the spinning threshold of the epoch with smallest inefficiency

number of spinning VCPU preemptions. For example, in our experiments, an epoch corresponds to 1000 preemptions of spinning VCPUs. Actual epoch lengths vary for different workloads. When the VM rarely uses spin-locks or the server is under-subscribed, spinning VCPU preemptions are rare, and thus epochs are long time intervals; when the VM is spinlock-intensive and is competing for CPU resources with other VMs, spinning VCPU preemptions are frequent, and thus epochs are short time intervals. With short epochs, APLE can quickly respond to execution phase changes. With long epochs, APLE can minimize runtime overhead. At the same time, this way of setting epoch lengths also guarantees that there are enough sample events in each epoch so that the *inefficiency* can be reliably calculated.

3.2.2 Heuristic VCPU Scheduling (HVS)

The selection of candidate VCPUs has direct impact on performance. Excessive spinning is usually caused by waiting for preempted VCPUs. As explained in Section 3.1, for the best performance, these VCPUs should be rescheduled as quickly as possible to avoid additional spinning on the VCPUs that are currently waiting for them or may wait for them in the future before they are rescheduled. Specifically, if excessive spinning is caused by the LHP problem, the VCPU holding the spin-lock should be selected and rescheduled first; if excessive spinning is caused by the LWP problem, the VCPU waiting at the beginning of the ticket-lock queue should be rescheduled first. However, due to the semantic gap between the VMM and VMs, the VMM does not have information to diagnose the root causes of the excessive spinning or distinguish such VCPUs from other preempted VCPUs. This make VCPU selection a challenging problem.

This chapter proposes a Heuristic-based VCPU Scheduling (HVS) algorithm to address candidate VCPU selection problem. The HVS algorithm assumes that the spinning thresholds have been appropriately set. Thus, spinning VCPUs will not be

preempted prematurely. While HVS can be implemented to work independently, it achieves better performance when utilized together with APLE, as we will show in Section 3.3.

The basic idea behind HVS is to evaluate and rank VCPUs based on the possibility and effectiveness to reduce spinning if they are rescheduled immediately. Similar to the methods in KVM, we first categorize "ready" VCPUs into two categories. *Resource-waiter VCPUs* are those preempted because of the depletion of their time slices and are waiting for CPU resources to resume execution; and *lock-waiter VCPUs* are those waiting for a spin-lock and preempted because of excessive spinning. A natural reason for such categorization is that resource-waiter VCPUs are ready to make progress and rescheduling them before lock-waiter VCPUs causes less spinning. A more important reason is that HVS needs to rank and schedule VCPUs in these two categories in different ways, as we will explain below.

HVS ranks the VCPUs in the same VM based on two heuristics. One is the *causality heuristic*, which schedules resource-waiter VCPUs before lock-waiter VCPUs. The rationale of the heuristic is that, when there are VCPUs preempted due to excessive spinning, they are directly or indirectly waiting for other VCPUs that have been preempted due to the depletion of time slices (i.e., resource-waiting VCPUs).

In a LHP problem, a spinning VCPU is preempted when it is waiting for the preempted lock holder, which can be found in the resource-waiter category. The cases with spin-lock holders spinning in critical sections are rare.

When the spinning-suppression hardware facilities are used, the LWP problem becomes more complex. As shown in Figure 3.4, in a LWP problem, a ticket-lock waiter may be preempted in a few scenarios. First, a ticket-lock waiter is preempted because it ran out of its time slice. In this case, the ticket-lock waiter can be found in the resource-waiter category. Second, a ticket-lock holder has been preempted, and

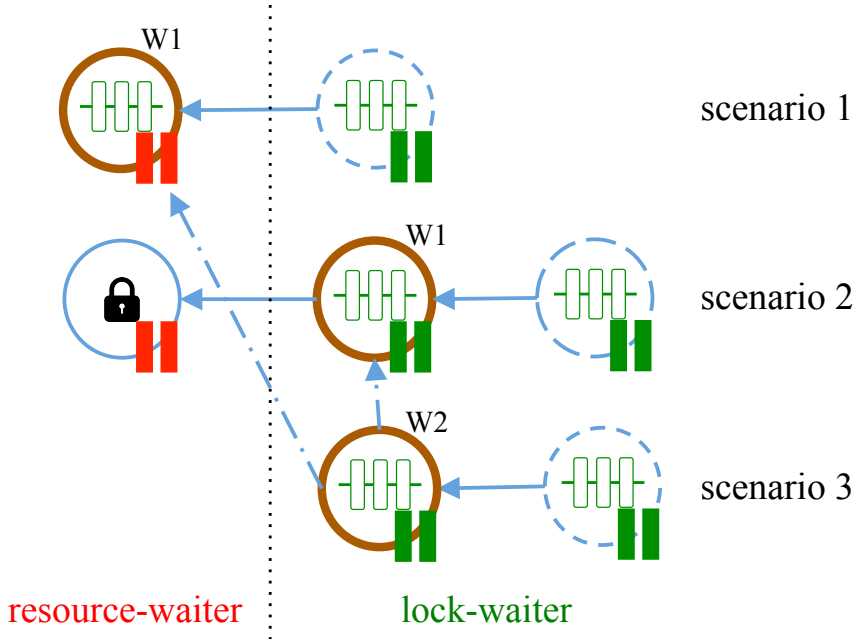

Figure 3.4 Three different scenarios of the LWP problem. The preempted ticket-lock waiter in each scenario is illustrated using a solid circle in thick line. A "pause" symbol in red color indicates that the corresponding VCPU is preempted due to the depletion of its time slice, and a "pause" symbol in green color indicates that the corresponding VCPU is preempted due to excessive spinning.

thus the ticket-lock waiter spins before it is preempted due to excessive spinning. In this case, the ticket-lock holder must be scheduled first, which is in the resource-waiter category. The ticket-lock waiter is in the lock-waiter category. Third, a ticket-lock waiter $W2$ is preempted because another ticket-lock waiter $W1$ located before it in the queue has been preempted. In this case, $W1$ should be scheduled before $W2$. $W1$ is in the resource-waiter category (as in the first scenario), or is in the lock-waiter category waiting for another VCPU in the resource-waiter category (as that in the second scenario). $W2$ is in the lock-waiter category.

Based on the analysis above, no matter whether the excessive spinning problem is caused by preempted lock holder or preempted ticket-lock waiter, a VCPU in the resource-waiter category should be scheduled before the VCPUs waiting for it in the lock-waiter category are scheduled. However, due to the semantic gap between the VMM and VMs, the VMM cannot identify which resource-waiter VCPUs are

blocking other VCPUs from making progress. Thus, a safe choice is to schedule all the resource-waiter VCPUs before scheduling the lock-waiter VCPUs.

The other heuristic, *preemption-time heuristic*, is used to rank the VCPUs in each category. When a VCPU is preempted, it is time-stamped. The timestamps keep increasing. HVS ranks resource-waiter VCPUs with larger preemption timestamps before the ones with smaller timestamps; and ranks lock-waiter VCPUs with smaller preemption timestamps before the ones with larger timestamps. This heuristic is based on the following observations.

When a VCPU (A) is preempted due to excessive spinning and resource-waiter VCPUs are examined, the resource-waiter VCPU (B) causing A to spin is more likely to be the one that is preempted recently. Critical sections and normal spinning in spin-lock synchronizations are much shorter than time slices. They are usually shorter than a few microseconds, while time slices are longer than a few milliseconds. Thus, the chance that spin-lock holders or spin-lock waiters are preempted due to depleted time slices is small if spin-locks are not frequently requested; and LHP and LWP problems are usually incurred by the workloads with frequent spin-lock synchronizations; for example, each VCPU many request a spin-lock multiple times in a time slice. Therefore, when A is preempted, B must have been preempted recently, later than the time when last time A requests the lock.

When all the "ready" VCPUs are lock-waiter VCPUs and there is still a VCPU being preempted due to spinning, the VCPU must be waiting for another VCPU, which shares the same ticket-lock with it and has been preempted earlier due to spinning. This corresponds to the third scenario in the LWP. Because all the VCPUs in the same VM use the same spinning threshold, the order in which the VCPUs are preempted by the hardware facilities reflects the order in which they request the ticket lock, which in turn determines their positions in the queue. Thus, these VCPUs should be scheduled in the same order as they are preempted.

Based on these two heuristics, the HVS maintains two lists, resource-waiter list and lock-waiter list, to organize resource-waiter VCPUs and lock-waiter VCPUs, respectively. HVS ranks the VCPUs on each list based on their preemption timestamps and ranks the resource-waiter VCPUs higher than lock-waiter VCPUs. When a VCPU needs to be selected, it just selects the VCPU with the highest rank[4].

3.2.3 APPLES Implementation

We have implemented APPLES based on KVM and Linux. The implementation of APLE in KVM adds only about 80 lines of source code to 4 existing files, and the implementation of HVS adds about 30 lines of source code to one existing file. Most changes are made inside the PLE event handler of KVM. Other changes are mainly to collect event times and other needed information (e.g., the preemption time of VCPUs, the number of times that the VCPUs have been preempted due to excessive spinning in each epoch, etc).

Every time when the spinning-suppression hardware detects excessive VCPU spinning, the PLE event handler is called to handle this issue. Inside the handler, APPLES first uses HVS to select a candidate VCPU. Then, it checks whether an epoch is finished or not. If an epoch is finished, it adjusts the spinning threshold and changes the Virtual Machine Control Structure (VMCS) of the VCPU accordingly, before it schedules in the VCPU.

One issue we addressed in the implementation is to adapt HVS to the method currently used in KVM to reschedule VCPU candidates. Linux and KVM uses virtual run time to schedule VCPUs. When a VCPU runs, its virtual run time increases monotonically. When the virtual run time exceeds any other VCPU's virtual run time by a time quantum (usually very small), the VCPU is preempted. In KVM,

[4]Though preferentially scheduling spinning VCPUs with larger preemption timestamps degrades performance (as shown in Figure 3.13), the "misclassification" of spinning VCPUs as resource-waiter VCPUs hardly hurts performance, due to its low possibility of happening.

when a spinning VCPU (A) is preempted and another VCPU (B) is selected, it uses a "yield_to" mechanism to temporarily boost the priority of B, such that B can be rescheduled as soon as possible. The virtual run time of B still keeps increasing. In HVS, the latest preempted VCPU is selected first. Since the latest preempted VCPU already has a large virtual run time (larger than that of any other VCPUs when it is preempted). Thus, it may be preempted again shortly after it is rescheduled. Then, it may be selected again by HVS when another spinning VCPU is preempted, though it is not blocking the progress of other VCPUs. This forms a loop preventing HVS from selecting VCPUs that can effectively reduce spinning. In the loop, a VCPU is selected by HVS repeatedly as a candidate VCPU. Since its virtual run time is large and keeps increasing, it is preempted shortly every time when it is scheduled, giving it a higher probability of being selected again by HVS. This not only lowers the quality of the candidate VCPUs selected by HVS, but also reduces the chances of other VCPUs getting rescheduled, and may cause starvation problem in the worst case.

To address this issue, the implementation in KVM prevents a VCPU from being selected as a candidate VCPU repeatedly. For this purpose, the implementation marks a VCPU as "yielded" when it is selected as a candidate VCPU. When a VCPU is preempted because its virtual run time is too large, its "yielded" mark is checked. If there is not a "yielded" mark, the VCPU is put onto the resource-waiter list. Otherwise, the mark is removed, and the VCPU is put onto a "yielded" VCPU list. The VCPUs on the "yielded" list are selected by HVS as candidate VCPUs when the resource-waiter list and lock-waiter list are empty. They may also be selected to run when their virtual run time is surpassed by that of other VCPUs.

3.3 Evaluation

This section evaluates APPLES with a collection of multi-threaded benchmarks. We first present the overall performance of APPLES. Then, for each component in APPLES, we carry out experiments to show its performance advantage and study in detail how it improves performance.

3.3.1 Experimental Setup

We conducted our experiments on a Dell PowerEdge R720 server with 64GB of DRAM and two 2.40GHz Intel Xeon E5-2665 processors. Each processor has 8 cores. There are 16 cores in total. On the server, we created 4 VMs with 16 VCPUs. Each VM has 16GB of memory. The VMM is KVM [10]. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. The VCPUs in each VM are one-to-one pinned to physical cores. Our experiments show that the benchmarks achieve better performance under this configuration than they do without pinning the VCPUs. CPU power management can reduce the performance of the applications running in VMs [11]. To prevent such performance degradation, in the experiments, we disabled the C states other than C0 and C1 of the processors, which have long switching latencies.

To evaluate APPLES, we used the benchmarks in PARSEC 3.0 suite [12], including native PARSEC benchmarks and SPLASH2X benchmarks in the suite. We attach a prefix 'p.' before the name of each native PARSEC benchmark, and attach a prefix 's.' before the name of each SPLASH2X benchmark, in order to differentiate these two sets of benchmarks. We also refer to native PARSEC benchmarks as PARSEC benchmarks for brevity. These benchmarks are mainly for testing multicore processor designs in computer architecture area. Most of them are computation-intensive and require minimal system support. Therefore, we also selected a few other applications that have been frequently used to study LHP and

LWP problems. *Ebizzy* [45] is multi-threaded and generates workloads similar to those on common web application servers. *Dbench* [46] is derived from an industry-standard benchmark *NetBench*. It is a utility that tests the ability of a file system to service requests from clients. *Hackbench* [47] is a multi-threaded benchmark designed to test Unix-socket (or pipe) performance. *Kernbench* [48] is a CPU and memory intensive benchmark that measures and compares the time used to compile Linux kernels. We selected these applications not only because they incur frequent system operations, but also because they are representative workloads in diverse application domains.

We compiled the PARSEC and SPLASH2X benchmarks using gcc with the default settings of the *gcc-pthreads* configuration in PARSEC 3.0. We built other benchmarks using the make files/scripts coming with the benchmark packages. The gcc compiler and the libraries required by the benchmarks are stock software components in the Ubuntu Linux distribution. We used the *parsecmgmt* tool in the PARSEC package to run the PARSEC and SPLASH2X benchmarks with native input. In the experiments, we set the number of threads in each benchmark equal to 32. We ran each experiment five times and report the average result.

We ran the benchmarks using the default KVM configuration and use their performance as the baseline performance. Since different benchmarks may use different metrics (e.g., throughputs and execution times) and the absolute performance numbers vary widely across benchmarks, we normalize the performance measured in the experiments against the baseline performance. Thus, the baseline performance is always 1. To be consistent, we use large values to represent higher performance. Thus, if a benchmark reports throughput, we present its normalized throughput. If a benchmark reports execution time, we present its speedup. For brevity, we use "performance" to refer to both normalized throughput and speedup.

The LHP and LWP problems happen when PCPUs are over-subscribed. Thus, we launch multiple VMs. We run multiple instances of the same benchmark in parallel

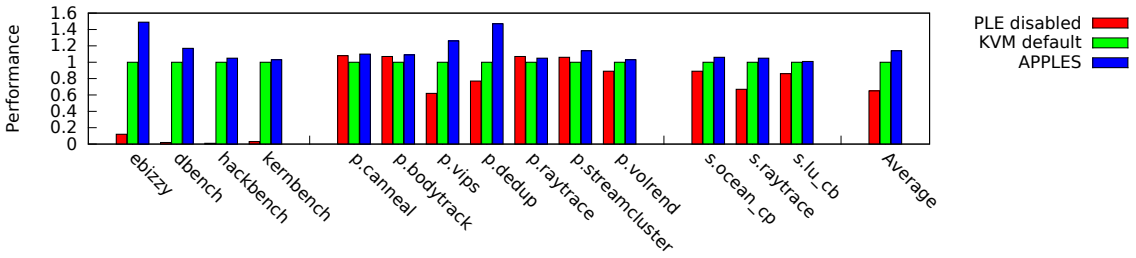

Figure 3.5 Normalized performance of the spinlock-intensive benchmarks with *KVM* and *APPLES* (PLE support enabled) and PLE support disabled, when 2 VMs co-run. Prefixes 'p.' in benchmark names stand for PARSEC benchmarks, and prefixes 's.' stand for SPLASH2X benchmarks.

on the VMs, one on each VM. different configurations, and compare the performance. In our experiments with APLE enabled, for all the VMs, the initial value of the desired threshold T_d is 8192 cycles. The lower bound T_l is 4096 cycles (the same as that in the default KVM setting). The upper bound T_u is 32768 cycles, and δ is 1024 cycles.

3.3.2 Overall Performance of APPLES

In this subsection, we first show the performance advantage of APPLES over the stock KVM for spinlock-intensive benchmarks. Then, we compare the overhead of APPLES and the stock KVM.

For each benchmark, we launch two VMs and run two instances of the benchmark in parallel, one on each VM. We first run the benchmark using the stock KVM with PLE disabled. Then, we enable the PLE support, and run the benchmark with the stock KVM and APPLES, respectively. We also manually set the PLE_window to 512 cycles, collect inefficiency values during its execution, and average the inefficiency values. If the average inefficiency value is greater than 5%, the benchmark is considered to be spinlock-intensive. We use spinlock-intensive benchmarks to evaluate the effectiveness of APPLES and non-spinlock-intensive benchmarks to test the overhead of APPLES.

Figure 3.5 shows the performance of spinlock-intensive benchmarks and their average performance under three scenarios, i.e., (1) with the PLE support turned off, (2) with the stock KVM (PLE enabled), and (3) with APPLES (PLE enabled). APPLES performs consistently better than the stock KVM for these benchmarks. Compared to the stock KVM, APPLES improves the performance of the benchmarks by 14% on average and up to 49%.

Among these benchmarks, *ebizzy*, *dbench*, *hackbench*, and *kernbench* incur the most frequent spin-lock operations. Their performance suffers significantly from LHP and LWP problems. Though the PLE support in the stock KVM can significantly improve their performance, APPLES is more effective and can further improve performance. For *p.canneal*, *p.bodytrack*, *p.raytrace*, and *p.streamcluster*, with the stock KVM, enabling PLE even degrades their performance, because the stock KVM cannot set spinning thresholds adequately and preempts spinning VCPUs prematurely. APPLES can avoid this problem. It achieves similar (for *p.raytrace*) or higher (for the other three benchmarks) performance, relative to that with the PLE support turned off. For the remaining benchmarks, the spin-lock operations in their executions are not as frequent as those in the first four benchmarks. With PLE support, the stock KVM improves their performance moderately, and APPLES can improve the performance by larger percentages.

APPLES improves performance through the synergistic collaboration of APLE and HVS, which significantly reduces the total cost incurred by excessive spinning and preempting spinning VCPUs. We use *ebizzy* as an example to illustrate how APPLES with its components reduces the cost and how the performance is affected by the reduction of the cost. To test one component of APPLES, we disable the other component and use the default mechanism in KVM.

As shown in Figure 3.6, compared to the stock KVM, the performance of *ebizzy* is improved by 34% with APLE alone, and is improved by 9% with HVS alone.

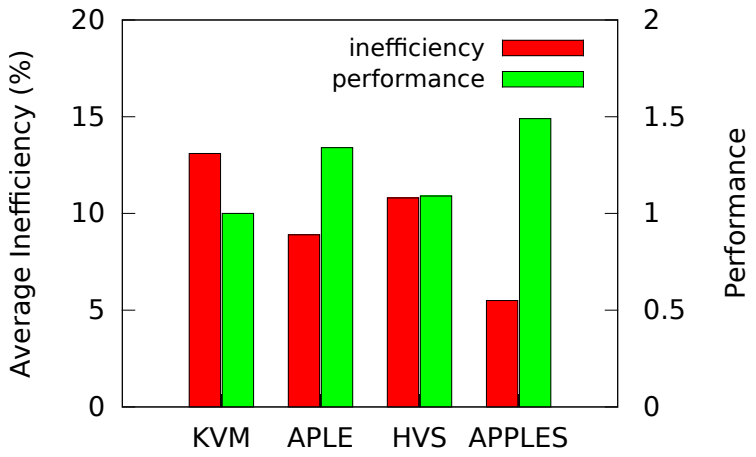

Figure 3.6 Normalized performance and average inefficiency of *ebizzy* with *KVM*, *APLE*, *HVS*, and *APPLES* when 2 VMs co-run

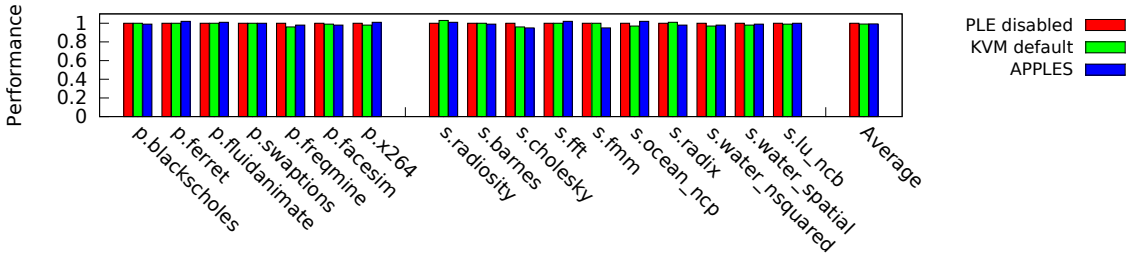

Figure 3.7 Normalized performance of the non-spinlock-intensive benchmarks with *KVM* and *APPLES* (PLE support enabled) and PLE support disabled, when 2 VMs co-run. Prefixes 'p.' in benchmark names stand for PARSEC benchmarks, and prefixes 's.' stand for SPLASH2X benchmarks.

With APLE and HVS combined, the performance can be improved by 49%. The percentage of improvement with APPLES is even higher than the sum of percentages of improvement with APLE and HVS alone. This is because HVS is more effective with APLE than it with the default mechanism in KVM to adjust spinning threshold, as we will show later in subsection 3.3.4. The figure also compares the average inefficiency values of *ebizzy* executions under these scenarios. APLE and HVS can reduce inefficiency by 32% and 18% respectively, and reduce it by 58% when combined, relative to the stock KVM. It is evident that performance is improved when the inefficiency decreases.

We measure the overhead of APPLES with two sets of experiments. The first set studies its overhead on under-subscribed systems. For this purpose, we launch one VM, and run spinlock-intensive benchmarks in the VM. On a system that is under-subscribed, each VCPU gets a dedicated physical core. Thus, lock holder/waiter VCPUs would not be preempted, and there is no need to preempt spinning VCPUs. Descheduling and rescheduling spinning VCPUs degrades performance. Thus, the performance measured with the PLE support disabled represents the best performance these benchmarks can achieve. The performance degradation caused by KVM and APPLES enabling and handling PLE events represents their overhead. On average, the benchmarks achieve similar performance with APPLES and the stock KVM, and the performance difference is not noticeable (less than 2%). Compared to the executions with PLE support disabled, these benchmarks show only slightly lower performance with KVM and APPLES (1%∼2% on average and up to 8% for *kernbench*). The overhead of APPLES is similar to that of the stock KVM and is acceptable when the system is under-subscribed.

The second set of experiments study the overhead of APPLES on over-subscribed systems. We launch two VMs, on which we run the benchmarks that do not incur frequent spinlock operations. Figure 3.7 shows the performance of these benchmarks and their average performance. We use performance tested with the stock KVM with PLE support disabled as baseline performance. Both APPLES and the stock KVM show similar performance as that with PLE support disabled (difference<1%), indicating that their overhead is very low for the benchmarks that rarely incur spinlock operations.

3.3.3 APLE Performance

To study in detail how APLE improves system performance, we enable APLE and disable HVS in APPLES. We select seven spinlock-intensive applications for the

study. We select *ebizzy*, *dbench*, *hackbench*, and *kernbench*, because they are more spinlock-intensive than other benchmarks, and their performance is more sensitive to the management of PLE facility. We select *p.raytrace* and *p.streamcluster*, because we want to investigate the reasons why APPLES can maintain and improve the performance while the stock KVM degrades their performance when PLE support is enabled. Benchmark *p.dedup* is selected because its performance is most sensitive to the management of PLE facility among the remaining benchmarks, which are not as spinlock-intensive as the first four benchmarks.

We carry out experiments to compare APLE against the mechanism which uses a fixed system-wide spinning threshold. Since a benchmark shows different performances with different spinning thresholds, we repeat experiments and test different spinning thresholds from 512 cycles to 32768 cycles to get a scope of performance variation. Thus, we can find the ***"best"*** performance and the ***"worst"*** performance that the benchmark can achieve by selecting different ***fixed*** spinning thresholds. In this section, we use *"best"* to represent the case in which the selected spinning threshold leads to the best performance, and use *"worst"* to represent the case in which the selected spinning threshold leads to the worst performance.

Please note that the "best" and "worst" performances are only those achieved with fixed spinning thresholds. They do not represent the real best and worst performance that can be achieved with any possible methods. However, we use the "best" performance and the "worst" performance to show the potential of adjusting the spinning threshold and how much performance degradation could be caused if the spinning threshold was not adequately set.

We also want to compare the "best" and "worst" performances against the performance that can be achieved with the dynamic method in APLE, and show the necessity for adjusting the spinning threshold dynamically based on workloads. During the execution of a benchmark, there may be different phases. A threshold

leading to good performance in one phase may lead to bad performance in another phase. Thus, it is possible that, with a spinning threshold adjusted dynamically, a benchmark achieves better/worse performance than the "best"/"worst" performance achieved with a fixed threshold used across different phases.

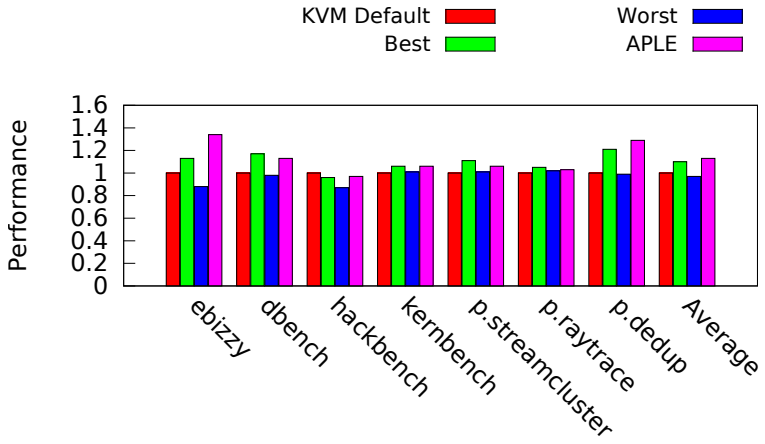

Figure 3.8 Normalized performance of the benchmarks with *KVM*, *"best"*, *"worst"*, and APLE when 2 VMs co-run.

Figure 3.8 shows the performance of these benchmarks when 2 VMs co-run. The stock KVM cannot achieve the best performance. Especially, with *p.streamcluster*, *kernbench* and *p.raytrace*, it even achieves lower performance than the "worst" performance obtained with a fixed spinning threshold level. In contrast, APLE can achieve better performance than *"best"* — the best performance that can be obtained by smartly selecting a fixed spinning threshold. The average performance achieved with APLE is 1.13, and the average performance achieved by smartly selecting a fixed spinning threshold (i.e., *"best"*) is 1.10. APLE improves the performance of *ebizzy* by the largest percentage (34% relative to the stock KVM and 19% relative to *"best"*). For *p.raytrace* and *p.streamcluster*, the "best" performance is achieved when the thresholds are high (32768 cycles). The stock KVM degrades performance because it sets the thresholds too low, such that spinning VCPUs are preempted prematurely. APPLES avoids this problem since premature VCPU preemptions increase wasteful

VCPU switches and thus inefficiency. The figure also shows that, when selecting a wrong spinning threshold level, the performance can be degraded by 16% on average and up to 46% (for *ebizzy*), relative to that with spinning thresholds adequately set by APLE.

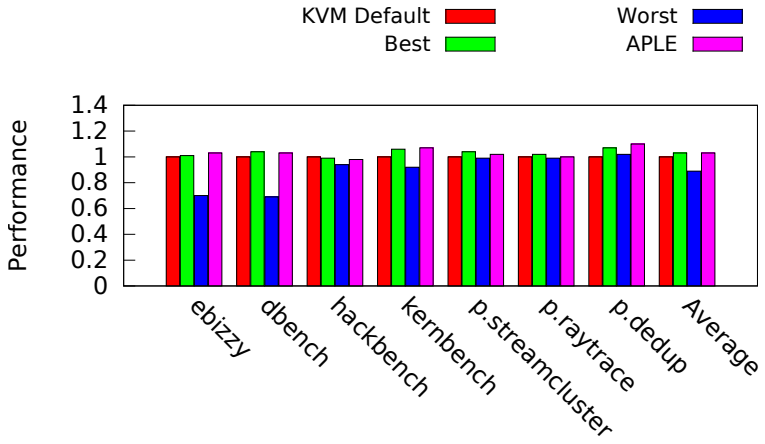

Figure 3.9 Normalized performance of the benchmarks with *KVM*, *"best"* and *"worst"*, and *APLE* when 4 VMs co-run.

Figure 3.9 shows the performance of the benchmarks when 4 VMs co-run. Compared to the executions with 2 VMs, the performance difference between the stock KVM, *"best"*, and APLE is much smaller. However, if the spinning thresholds are set inadequately, application performance still can be significantly reduced. For example, with *dbench*, the performance difference between *"best"* and *"worst"* is 19% when 2 VMs co-run, and the difference is increased to 35% when 4 VMs co-run.

To illustrate the correlation between system performance and the inefficiency level and to show how adjusting spinning threshold can reduce inefficiency and improve system performance, we use *ebizzy* as an example, and compare the average inefficiency values along with normalized performances achieved by KVM, *"best"*, *"worst"*, and APLE when 2 VMs co-run. The average inefficiency is the average

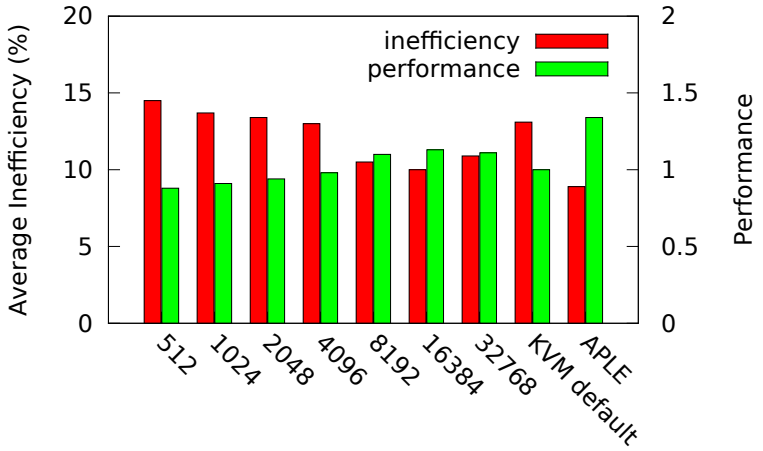

Figure 3.10 Normalized performance and average inefficiency of *ebizzy* when a system-wide spinning threshold is changed from 512 cycles to 32768 cycles, and when the stock *KVM* and APLE is used to adjust the spinning threshold. Two VMs are used.

of the inefficiency values measured in the epochs of the two VMs during the two instances of *ebizzy* run in parallel in the VMs.

As shown in Figure 3.10, in general the average inefficiency reduces when the spinning threshold is increased from 512 cycles to 16384 cycles. This is because the overhead of wasteful VCPU switches caused by preempting spinning VCPUs prematurely can be reduced with larger spinning thresholds. Meanwhile, with the decreasing of the average inefficiency, the performance is improved accordingly. However, when the spinning threshold is further increased, the average inefficiency increases, since the overhead of wasteful spinning starts to dominate, and thus the performance is degraded.

Figure 3.10 also clearly shows that, with a fixed spinning threshold, the "best" performance is achieved when the average inefficiency is minimized by smartly selecting the spinning threshold (16384 cycles in this case). The default KVM mechanism cannot achieve the best performance since it cannot effectively reduce inefficiency. In contrast, APLE reduces the average inefficiency by 32%, relative to the stock KVM. Moreover, compared to *"best"*, APLE reduces the average inefficiency

by 11%, which is the reason why APLE can achieve even higher performance than *"best"*.

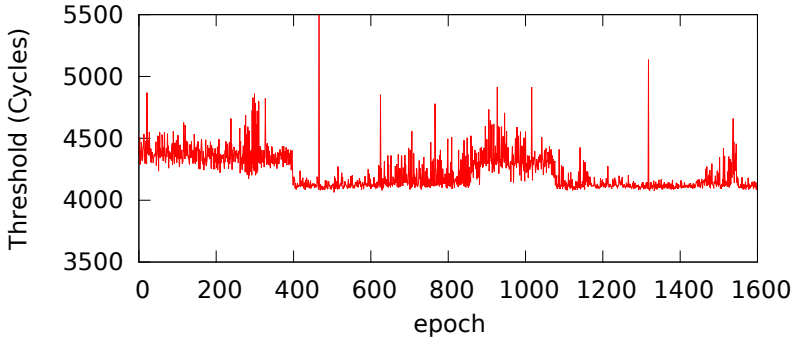

Figure 3.11 Spinning threshold adjusted by the stock KVM when two VMs co-run

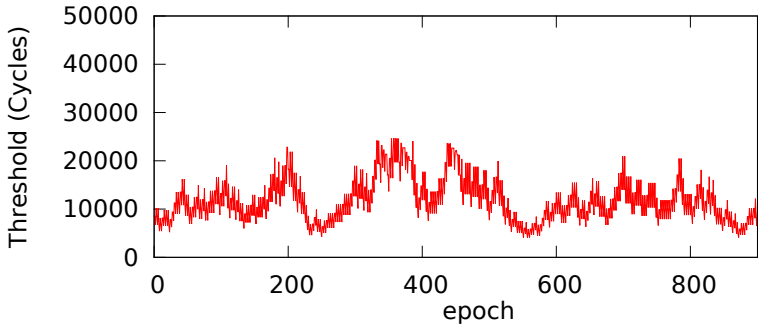

Figure 3.12 Spinning threshold adjusted by APLE when two VMs co-run

In the above experiments, we also collected the spinning thresholds during the execution of the *ebizzy* instances[5]. Figures 3.11 and 3.12 show how spinning thresholds are adjusted respectively for the scenarios with default KVM mechanism and APLE. With APLE, there are about 900 epochs in the execution, while with KVM default mechanism there are about 1600 epochs. This is because fewer VM_EXITs are incurred by PLE events with APLE. With the default KVM mechanism, the spinning threshold sticks round 4200, which leads to the poor performance similar to the one with fixed spinning threshold of 4096. This shows that the stock KVM cannot effectively adjust the spinning threshold to achieve optimal performance. However, with APLE, the spinning threshold changes steadily around 12000, which leads to the

performance that is even better than "best" performance achieved with fixed spinning threshold of 16384.

3.3.4 HVS Performance

In this section, we want to understand how the heuristics in HVS help improve the performance. For this purpose, we have implemented three variants of HVS, which intentionally avoid selecting the candidate VCPUs suggested by a heuristic. The name of the variants and their differences with HVS are:

- CPTH-R (Counter Preemption-Time Heuristic on Resource-waiters): when selecting a candidate VCPU from resource-waiter VCPUs, the one with the smallest preemption timestamp is selected.
- CPTH-L (Counter Preemption-Time Heuristic on Lock-waiters): when selecting a candidate VCPU from lock-waiter VCPUs, the one with the largest preemption timestamp is selected.
- CCH (Counter Causality Heuristic): lock-waiter VCPUs are selected before resource-waiter VCPUs.

We select the same set of benchmarks as we do for testing APLE, and compare the performance of HVS with its variants when we run the benchmarks in 2 VMs. Figure 3.13 and Figure 3.14 show their performance, relative to the stock KVM. In Figure 3.13, the data was obtained with the default mechanism in KVM to adjusting spinning threshold. In Figure 3.14, the data was obtained with APLE adjusting spinning thresholds.

As shown in Figure 3.13, HVS performs slightly better than its variants when the default mechanism in KVM adjusting spinning threshold. The average performance is 1.08, 1.05, 0.97, and 1.03 for HVS, CPTH-R, CPTH-L, and CCH, respectively.

[5]The default KVM mechanism does not use epochs and sets a spinning threshold for each VCPU. For fair comparison, we define epoch in the same way as in APLE (i.e., 1000 VM_EXITs caused by PLE events), and collect the average spinning threshold of all the VCPUs in a VM for each epoch.

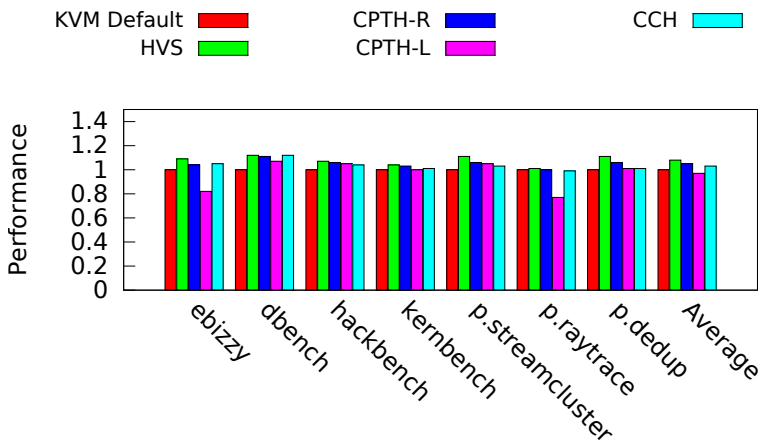

Figure 3.13 Normalized performance of the benchmarks with the stock KVM, HVS, and three variants of HVS when 2 VMs co-run. The default mechanism in KVM is used to adjust spinning thresholds.

This indicates that the heuristics used in HVS do help improving the performance, but they are not very effective. The figure also shows that the preemption-time heuristic applied to lock-waiter VCPUs helps improving performance by the largest percentage. This is because, for ticket spin-locks, the order in which lock-waiter VCPUs are scheduled has great impact on performance.

We were surprised to see that the causality heuristic is not as effective as the preemption-time heuristic. The benchmark *dbench* even shows the same performance on HVS and CCH. Our investigation shows that the default mechanism in KVM tends to adjust the spinning thresholds to very low values (Figure 3.11 and Figure 3.12), which preempt spinning VCPUs prematurely even when LHP or LWP does not happen. Thus, scheduling resource-waiter VCPUs first and scheduling lock-waiter VCPUs first do not make much difference on performance. Note that lock-waiter VCPUs are preempted when their spinning thresholds are reached. Thus, if spinning thresholds are set too low, lock-waiter VCPUs are preempted even when they may get the locks shortly.

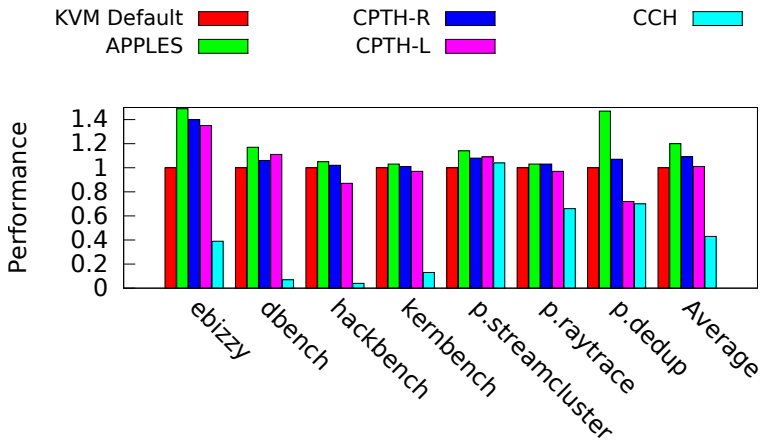

Figure 3.14 Normalized performance of the benchmarks with the stock KVM, HVS, and three variants of HVS when 2 VMs co-run. APLE is used to adjust spinning thresholds for HVS and its variants.

Thus, we repeated the experiments with APLE adjusting spinning thresholds of the VMs. As shown in Figure 3.14, the heuristics in HVS become more effective when spinning thresholds are adequately set. Not using these heuristics leads to serious performance degradation. The average performance is 1.20, 1.09, 1.01, and 0.43 for HVS, CPTH-R, CPTH-L, and CCH, respectively. Specifically for the causality heuristic, scheduling lock-waiter VCPUs before resource-waiter VCPUs reduces the performance of *dbench* and *hackbench* by more than 10x. This on one hand shows the importance of VCPU selection and confirms the effectiveness of causality heuristic, and on the other hand demonstrates the effectiveness of APLE on selecting adequate spinning threshold to accurately and promptly identify VCPUs waiting for preempted lock holders or preempted lock waiters.

3.4 Related Work

A large number of studies have been focused on the lock holder preemption (LHP) and the lock waiter preemption (LWP) problems. Various solutions have been proposed to reduce performance degradation. Software-only solutions include sophisticated VCPU scheduling algorithms [21, 49, 5, 6, 50, 22, 51, 52, 53, 54, 55], improved synchronization

primitives [24], and paravirtualization [56, 57]. On current platforms, using spinning-suppression hardware facilities, such as Intel PLE and AMD PF, has been dominantly utilized on mainstream virtualization systems and become a *de facto* standard solution [4, 16, 23, 25]. Our work does not provide an alternative solution to the LHP and LWP problem. Instead, it improves the solutions with hardware facilities.

Targeting the problem of setting spinning thresholds for the hardware facilities, there are studies showing that spinning thresholds must be adjusted based on workloads to achieve best performance [58, 44]. Besides APLE [59], there are some other efforts to adjust spinning thresholds dynamically. Zhang, Dong, and Duan [44] proposed a profiling method that collects the average spin-lock cycles in guest OSs and uses the information to adjust spinning thresholds. This approach requires the VMM to have detailed and important information about guest OSs, such as OS symbol tables, which should not be exposed to the VMM for security reasons on the systems shared by multiple users, e.g., public clouds. This seriously limits the scope of the solution. Thimmappa [40] proposed a method to adjust the spinning threshold based on whether or not the resources freed by preempting spinning VCPUs can be reallocated to other VCPUs for them to make progress. Recently, KVM implemented a method to dynamically grow/shrink the spinning threshold for each VCPU [41]. These two methods focus mainly on improving the performance when the system is under-committed.

Targeting the problem of which VCPUs should be scheduled to replace spinning VCPUs, besides the directed yield method currently used in KVM [43], and another Linux online patch [60], which relies on modified guest OSs to label VCPUs holding spin-locks, we cannot find any research on selecting VCPUs for the efficient utilization of spinning suppression hardware facilities.

The trade-off between busy waiting (spinning) and blocking in synchronization primitives is a classic yet challenging problem, and has been intensively studied

under different scenarios [61, 62, 63, 64]. The problem we target in this chapter also needs to make a trade-off between busy waiting and blocking. But, compared to the problems targeted in previous studies, the problem in this chapter is more challenging, since the VMM has limited information and cannot directly control the spinning in synchronization primitives.

3.5 Conclusion

Mainstream virtualization systems rely on hardware facilities, such as Intel PLE and AMD PF, to alleviate the performance degradation due to excessive VCPU spinning. However, it is still a challenging issue to effectively control these facilities to minimize overhead and maximize throughput, which requires the knowledge on the locking behaviors of guest systems that is unavailable at the VMM level, due to the semantic gap between the host and the guests. Ineffective utilization of these hardware facilities may even cause performance degradation.

This chapter addresses this issue with a holistic solution named APPLES. The two components in it solve two core problems in the utilization of the hardware facilities. Specifically, one component APLE maintains an adequate VCPU spinning threshold for each VM, in order to promptly detect and preempt VCPUs when they spin excessively. The key idea is to measure the execution efficiency of each VM and adjust the threshold in a way to maximize the efficiency. The other component HVS carefully selects VCPUs and schedules them in an order required by efficient synchronization. The key idea is to evaluate and rank VCPUs based on the causality and time of VCPU preemptions.

Our experiments show that APPLES can improve system performance by as much as 49%. Its implementation incurs minimal modification to existing virtualization system designs. We seek the adoption of APPLES in commercial and open-source virtualization systems.

CHAPTER 4

RETHINKING THE SCALABILITY OF MULTICORE APPLICATIONS ON BIG VIRTUAL MACHINES

4.1 Introduction

In the cloud, virtual machine (VM) sizes increase steadily to meet the demand for increasing computing power in each VM and to utilize the growing core counts in underlying physical machines. For example, a X1 instance on Amazon EC2 platform now has as many as 128 virtual CPUs (VCPUs) [65]. With increasing VM sizes, an important question to answer is how well applications can scale and take advantage of the computing power of bigger VMs to improve performance.

Common practice assumes that virtual machines have a similar architecture to their hosting physical machines (PMs), and thus the execution scalability of multicore applications on VMs can be analyzed in the same way as that on dedicated physical machines. As an evidence, VMs with multiple VCPUs on x86 architecture are called SMP-VMs or virtual SMPs [22], and Amdahl's law is used to analyze scalability.

However, due to the sharing of physical CPU resource on virtualized platforms and the dynamic CPU resource allocation for enabling the sharing, VCPUs show substantially different behaviors and performance features than physical computing cores. Thus, applications show different scalability on VMs than they do on physical machines. For example, research has shown that some multicore programs may suffer lower scalability on VMs, because the VCPUs in a VM may not make progress continuously and simultaneously [22, 26].

Although a few scalability problems have been noticed on VMs and the specific reasons have been analyzed, how the scalability of multicore applications in the cloud is changed by the virtualization of CPU resource has not been systematically studied. This chapter analyzes and verifies with experiments how CPU resource sharing in

virtualization impacts application scalability, identifies key application features and system factors affecting application scalability, and explore the potential and design alternatives for improving application scalability.

First, by re-defining speedup based on resource utilization efficiency as a measurement of scalability, this chapter analyzes and reveals the fundamental reasons for multicore applications showing different scalability on VMs than they do on PMs (Section 4.2). Second, based on the analysis, this chapter identifies two key application features for scalability and shows with experiments how virtualization affects scalability differently for the applications with different features (Section 4.3). Though applications are usually considered to have similar or lower scalability on VMs, this chapter shows that virtualization tends to improve scalability. However, the improved scalability might be offseted by frequent synchronization and long scheduling delay. Third, though some applications already show better scalability on VMs than they do on PMs with existing system design, this chapter shows that there is still much space to further improve scalability on VMs. Thus, this chapter identifies the system factors that can be leveraged for improved scalability. This chapter investigates two factors that have not been mentioned or studied before in other literatures — allocation period length and the matching between resource allocation and workload distribution. With experiments, this chapter demonstrates that improved scalability can be achieved by increasing allocation period length or matching resource allocation and workload distribution (Section 4.4).

4.2 Resource Sharing's Impact on Scalability

Due to the sharing of CPU resource, the methods and models developed to analyze the execution scalability of applications on dedicated hardware, e.g., Amdahl's law, cannot be used for understanding the execution scalability of applications on VMs. This section first introduces how CPU resource is managed and shared on virtualized

platforms. Then, it provides a new method which measures scalability based on resource utilization efficiency. With the method, it explains how resource sharing affects execution scalability on VMs.

4.2.1 Resource Sharing between VMs

On a physical machine hosting multiple VMs, a virtual machine monitor (VMM) is used to manage and dynamically allocate hardware resource to each VM. For CPU resource, a physical CPU (PCPU) is usually time-shared by multiple VCPUs. The VMM treats VCPUs as independent schedulable entities and allocates CPU time to them. Inside each virtual machine, threads are further scheduled onto VCPUs by the guest OS. Thus, by having multiple VCPUs in each VM, the threads in the VM can eventually run on multiple PCPUs, achieving higher performance than that with a single VCPU.

When allocating CPU time, the VMM first allocates CPU time to VMs based on their weights, and then distributes CPU time to VCPUs in each VM. As in typical OS implementations, to guarantee responsiveness, CPU time is usually allocated periodically to VCPUs as their timeslices in each period. For brevity, we refer to the period in which CPU time is distributed to VCPUs as an **Allocation Period**.

A VCPU consumes its timeslice when it runs on a core. For improved efficiency, a VCPU is descheduled when it stops making progress (e.g., when it becomes idle or busy-waiting[1]), and stops consuming timeslice. If a VCPU is not rescheduled for long time, it is possible that the VCPU cannot consume up its timeslice in an allocation period. In such a case, the VMM usually does not roll over the unused timeslice or a part of the unused timeslice to the next allocation period, in order to prevent a VCPU from accumulating too much timeslice and starving other VCPUs on the same core.

[1]Most multicore processors have equipped with mechanisms, such as Intel PLE and AMD PF, to detect and interrupt busy-waiting.

4.2.2 Efficiency-Based Scalability Measurement

To analyze scalability on VMs, we introduce a new method, which measures scalability base on the utilization efficiency of CPU resource. Specifically, the scalability of an application is determined by how efficiently the increased resource is utilized during the execution of the application. The higher the efficiency (i.e., less waste) is, the more the application can be accelerated, and the higher the scalability is.

Scalability is how much an application can be accelerated if allocated with more resource, with speedup being a measurement. For CPU resource, the speedup of the execution on N processing unit (PU, i.e., cores in PMs or VCPUs in VMs) against that on 1 processing unit is as follows.

$$\begin{aligned} \textit{Speedup} &= \frac{\text{execution speed on N PUs}}{\text{execution speed on 1 PU}} \\ &= \frac{\text{work finished on N PUs in an unit of time}}{\text{work finished on 1 PU in an unit of time}} \end{aligned}$$

Without loss of generality, we assumes that the amount of work finished is proportional to the CPU time utilized for effective computation. Since the total CPU time available on N PUs is N times of that on 1 PU, the above equation can be rewritten as follows.

$$\begin{aligned} &\textit{Speedup} \\ &= \frac{\text{total CPU time utilized on N PUs in an unit of time}}{\text{CPU time utilized on 1 PU in an unit of time}} \\ &= N \times \frac{\frac{\text{total CPU time utilized on N PUs in an unit of time}}{\text{total CPU time available on N PUs in an unit of time}}}{\frac{\text{CPU time utilized on 1 PU in an unit of time}}{\text{CPU time available on 1 PUs in an unit of time}}} \\ &= N \times \frac{\text{overall utilization efficiency with N PUs}}{\text{utilization efficiency with 1 PU}} \end{aligned}$$

In the above equation, **utilization efficiency is the ratio between the amount of utilized CPU time and the amount of available CPU time**, and

the utilized CPU time is that consumed for effective computation. The CPU time spent on busy-waiting does not count. Assume the utilization efficiency of 1 PU (i.e., serial execution) is 100%. The speedup with N PUs can be simplified as follows.

$$\begin{aligned} \textit{Speedup} &= N \times \text{overall utilization efficiency with N PUs} \\ &= N \times \frac{\text{CPU time utilized by computation}}{\text{available CPU time during computation}} \\ &= N - N \times \frac{\text{unutilized CPU time}}{\text{available CPU time during computation}} \end{aligned}$$

In the equation, *unutilized CPU time* refers to the CPU time that is not utilized by the application to make progress.

The above definition of speedup is consistent with that based on execution time. For example, based on Amdahl's law, if in an application 20% of computation can only be executed sequentially and 80% of computation can be fully parallelized without overhead, when executed on a 4-core machine, the speedup against the execution on a single core machine is $1/(0.2 + 0.8/4) = 2.5$. The performance does not scale linearly. This is because, when the sequential portion is executed on one core, other cores are idle. This reduces the utilization efficiency to 50% on these cores. The overall utilization of the 4 cores is $(100\% + 3 \times 50\%)/4 = 62.5\%$, and the speedup based on the above definition is $4 \times 0.625 = 2.5$.

The above definition of speedup can be used to understand both the scalability on physical machines with dedicated resources and the scalability on VMs with shared resources. To highlight the reasons causing different scalabilities on these platforms, we adapt the speedup calculation for physical machines and virtual machines respectively as follows.

For the executions on a physical machine, *unutilized CPU time* is the CPU time wasted on idling and busy-waiting during the execution. Thus, the speedup of an application on a physical machine with N cores can be calculated as follows.

$$\begin{aligned}\text{Speedup\_PM} &= N - N \times \frac{\text{time on idling and busy-waiting}}{N \times \text{execution time}} \\ &= N - \frac{\text{time on idling and busy-waiting}}{\text{execution time}}\end{aligned}$$

The CPU resource for a virtual machine is its timeslice. For the executions on a virtual machine, *unutilized CPU time* consists of two parts. The first part, *unused timeslice*, is the timeslice that cannot be depleted by the VCPUs in an allocation period and cannot be rolled over to later allocation periods (Section 4.2.1). The second part is the CPU time used to handle idle VCPUs and spinning VCPUs. Due to resource sharing, idleness and spinning are handled in a substantially different way on VMs than on PMs. To improve the utilization of shared CPU resource, hardware and the VMM usually try their best to detect and deschedule VCPUs that are not making progress, including idle VCPUs and spinning VCPUs. Thus, CPU time is not wasted on idling and busy-waiting. However, time must be spent to switch out these VCPUs. Therefore, the speedup of the execution on a VM with N VCPUs (against that on a VM with a single VCPU) can be calculated as follows[2].

$$\begin{aligned}\text{Speedup\_VM} = N - N \times & \\ & (\frac{\text{overhead of switching out idle/spinning VCPUs}}{\text{timeslice allocated to the VM}} \\ & + \frac{\text{unused timeslice of the VM}}{\text{timeslice allocated to the VM}})\end{aligned}$$

[2]With existing system designs, spinning that is very brief or at the user-level of VMs may not be detected. Such spinning still consumes CPU time. We choose to neglect the CPU time used by such spinning because 1) minimal CPU time is used by brief spinning and 2) excessive spinning at the user-level should be prevented using co-scheduling or interrupted using hardware facilities similar to Intel PLE and AMD PF.

4.2.3 Virtualization's Impact on Scalability

The impact of virtualization and CPU resource sharing on scalability can be identified by comparing the equations for calculating Speedup_PM and Speedup_VM. On a dedicated physical machine, resource provisioning is static. The scalability is mainly determined by the behavior of the application, i.e., whether the application can engage all the cores in useful work. Any idleness and busy-waiting are translated into lower resource utilization and then lower scalability. The reduction of scalability is proportional to the durations of idleness and busy-waiting. This also explains Amdahl's law and other models for analyzing execution scalability on dedicated hardware, which co-relate scalability with the operations causing idleness and busy-waiting, such as sequential computation, tasks on critical path, and synchronizations.

Virtualization affects scalability in two ways. On one hand, dynamic resource allocation helps improving scalability. For VCPUs, resource is not consumed if there is not useful work on them. Thus, even if an application cannot always engage the VCPUs in useful work, high scalability may still be achieved, as long as the overhead incurred by VCPU switches is low and most of the timeslice of the VM is eventually consumed by the end of resource allocation periods. An execution on VM may achieve linear scalability even if there is substantial sequential computation. Thus, conventional methods and models (e.g., Amdahl's law) become inapplicable when used to understand application scalability on VMs.

On the other hand, scalability on VMs is limited by new factors — VCPU switch overhead and unused timeslice of the VM. These factors are determined not only by the natures of the computation in applications but also the resource management at the system level. Specifically, VCPU switch overhead is proportional to the frequency of VCPU switches, which are usually incurred by the synchronizations on VCPUs. The more frequent the synchronizations are, the lower the scalability is. A few

factors affect the amount of unused timeslice. First, the amount of unused timeslice is determined by whether there is enough workload in each resource allocation period to consume timeslice. Second, timeslice and workload are distributed to each VCPU. Thus, the amount of unused timeslice is also determined by the distribution of workload and the distribution of timeslice to VCPUs. When a VCPU is allocated with more timeslice than needed by the workload on it, some timeslice will not be used. Finally, VCPU scheduling may also significantly affect the amount of unused timeslice. If a VCPU is scheduled late, the workload on it may not have enough time to consume the timeslice available to the VCPU by the end of a resource allocation period, increasing unused timeslice.

In Section 4.3, we identify and experimentally verify the application features affecting the scalability on VMs, and in Section 4.4, we investigate CPU resource management in the VMM and identify system-level factors affecting execution scalability.

4.3 Application Features Affecting Scalability

This section identifies two key application features for scalability and shows how these features affect application scalability on VMs.

4.3.1 Key Application Features and Scalability Indications

Based on the analysis in Section 4.2, we have identified two key scalability features of applications. One feature is *workload parallelism*, which describes to what degree an application can parallelize its workload in order to utilize increased CPU resource. During the execution of an application, its workload parallelism can be measured by the number of threads in the application that are active and making progress. In a time period, the higher the workload parallelism is, the more progress the application can make if provided with more resource. An application with higher workload

Table 4.1 Summary of Four Types of Applications Based on Their Key Scalability Features on VMs

Type	Work. Para.	Sync. Freq.	Scalability	Benchmarks
1	high	low	high	p.freqmine, p.swaption, p.x264, p.ferret, p.vips, s.water_nsquared, s.barnes, s.lu_ncb, s.raytrace, s.radix
2	low	high	low	p.bodytrack, p.dedup, p.facesim, s.ocean_cp, s.volrend, s.cholesky
3	high	high	mediocre	p.fluidanimate, p.streamcluster, s.ocean_ncp
4	low	low	mediocre	p.canneal, p.raytrace, p.blackschole, s.fmm, s.radiosity, s.water_spatial s.fft, s.lu_cb

parallelism tends to show higher execution scalability on both physical machines (because of less idle time) and virtual machines (because of less unused CPU time).

The other feature is *the frequency of blocking synchronizations* (referred to as *synchronization frequency* for brevity). Blocking synchronizations can incur the switches of VCPUs, which reduce scalability in two ways. First, when the threads on a VCPU are blocked, the VCPU is descheduled, and another VCPU (probably from another VM) is scheduled. The switch of VCPU incurs high overhead. Second, when a thread on the descheduled VCPU is unblocked and becomes ready to make progress, the VCPU may not be able to be rescheduled immediately. Due to this *rescheduling delay*, the VCPU may not be able to fully utilize the timeslice allocated to it by the end of allocation periods, increasing unutilized timeslice.

Based on these features, multicore applications can be categorized into four types, as summarized in Table 4.1 Applications with high workload parallelism and low synchronization frequencies (*type 1*) usually show high scalability on VMs; applications with low workload parallelism and high synchronization frequencies (*type 2*) usually show low scalability on VMs; applications with high workload parallelism and high synchronization frequencies (*type 3*) and applications with low workload parallelism and low synchronization frequencies (*type 4*) show mediocre scalability.

4.3.2 Experimental Verification

To verify the scalability indications of the aforementioned application features through experiments, we select the benchmarks in PARSEC 3.0 suite [12], including native

PARSEC benchmarks and SPLASH2X benchmarks. We attach a prefix 'p.' before the name of each native PARSEC benchmark, and attach a prefix 's.' before the name of each SPLASH2X benchmark, in order to differentiate these two sets of benchmarks. We also refer to native PARSEC benchmarks as PARSEC benchmarks for brevity. We used the parsecmgmt tool in the PARSEC package to run the benchmarks with native input and to control the number of concurrent threads in each execution.

Experiments were conducted a Dell PowerEdge R720 server with 64GB of DRAM and two 2.40GHz Intel Xeon E5-2665 processors. Each processor has 8 cores. On the server, we created 4 VMs. Each VM has 16GB of memory and 16 VCPUs. The VMM is KVM. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. The VCPUs in each VM were laid out on the cores in a way to prevent VCPU stacking for better performance [22].

We first profiled the benchmarks to obtain their scalability features when we run them on the physical server. The number of thread in each execution is 16. During the execution of each benchmark, we collected the number of active CPU cores involved in the benchmark computation periodically and the number of voluntary context switches[3]. The workload parallelism of the benchmark is the average number of active cores during its execution, and its synchronization frequency is the number of voluntary context switches per second.

Figure 4.1 shows the categorization of the benchmarks based on their scalability features. If a benchmark keeps at least 75% of the cores (i.e., 12 cores in our system) active on average during its execution, it is considered to have high workload parallelism. If the time period between two consecutive synchronizations is shorter than the timeslice allocated to a thread in an allocation period (i.e., a thread is

[3]Voluntary context switches are context switches caused by threads blocking their execution voluntarily, i.e., blocking synchronizations.

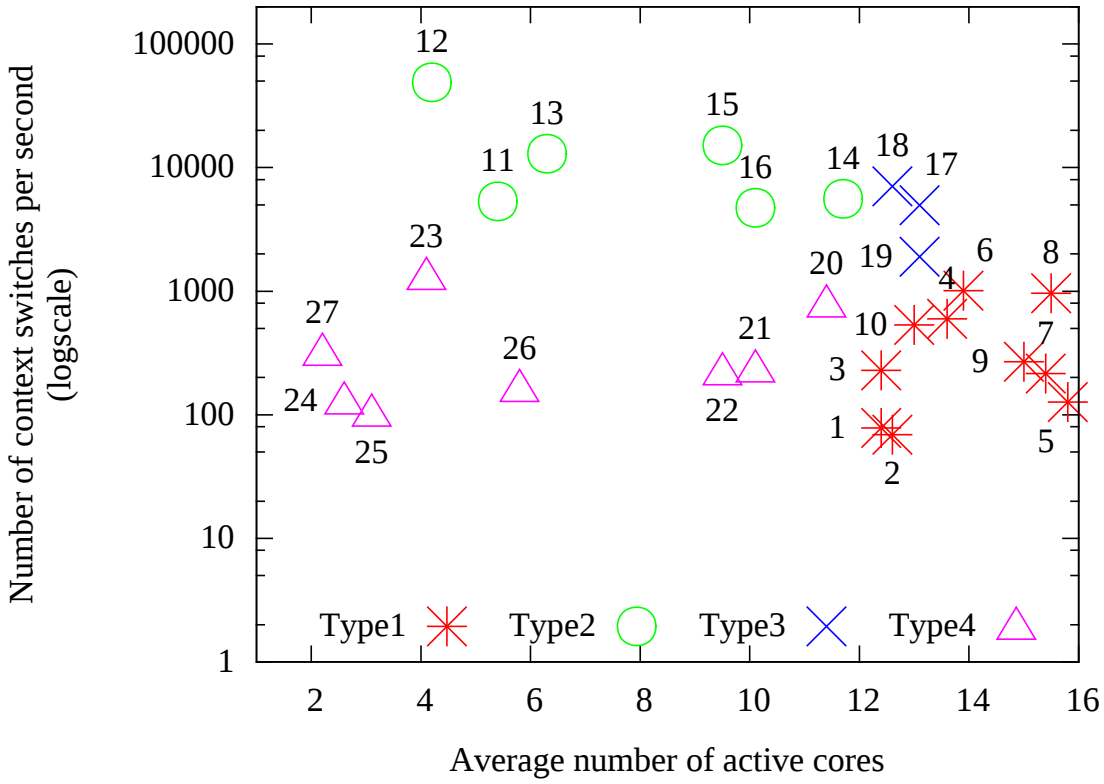

Figure 4.1 Types of the PARSEC benchmark and their scalability features. The numbers are the indexes of the benchmarks, which are indexed as follows. 1: p.freqmine, 2: s.water_nsquared, 3: s.barnes, 4: s.lu_ncb, 5: p.swaption, 6: p.x264, 7: p.ferret, 8: p.vips, 9: s.raytrace, 10: s.radix, 11: p.bodytrack, 12: p.dedup, 13: p.facesim, 14: s.ocean_cp, 15: s.volrend, 16: s.cholesky, 17: s.ocean_ncp, 18: p.streamcluster, 19: p.fluidanimate, 20: s.lu_cb, 21: s.water_spatial, 22: s.fmm, 23: p.canneal, 24: s.fft, 25: p.raytrace, 26: p.blackschole, 27: s.radiosity

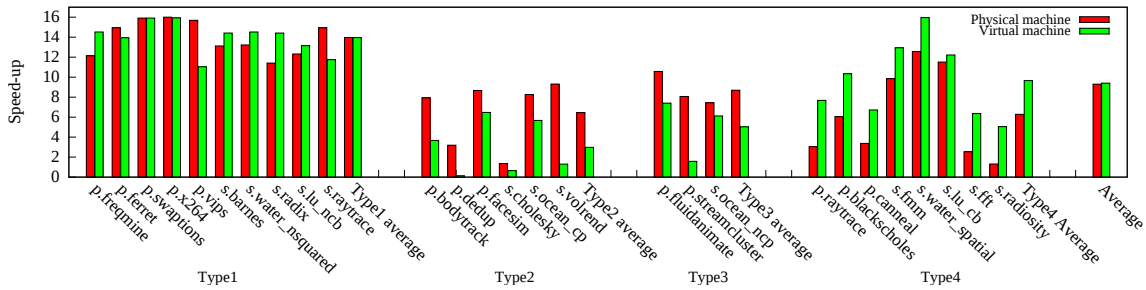

Figure 4.2 Speedups of PARSEC and SPLASH2X benchmarks.

blocked at least once before it uses up its timeslice), the benchmark is considered to have high synchronization frequency.

The benchmarks of each type are summarized in Table 4.1. Note that an application with low workload parallelism only means that the application lacks enough active threads to keep all the cores/VCPUs busy. As we will show later that an application with low workload parallelism may still achieve decent scalability on a VM. At the same time, the workload parallelism is relative to the scale of the system (e.g., the number of cores/VCPUs in a server/VM). An application with high workload parallelism may become one with low workload parallelism on a larger system.

Then, we run the benchmarks in a VM consolidated with three other VMs on the same physical server. To obtain stable measurement, we run a CPU-bound program in each of three VMs, which keeps increasing a counter on all the VCPUs of the VM. We show the speedups of the benchmarks in Figure 4.2. The concurrency level (i.e., the number of threads in the benchmark, the number of VCPUs in each VM, and the number of cores used in the PM) is 16. The speedup is relative to the performance with concurrency level equal to 1.

The benchmarks of the first type show the highest scalability, and the speedups are similar on the PM and the VM. The average speedups are both 13.9. Their high speedups are achieved for two reasons: 1) high workload parallelism ensures that

CPU resource is fully utilized; and 2) there are no factors reducing the utilization of CPU resource.

Other benchmarks show different scalability behaviors on the VM than on the PM. On the PM, the speedups are mainly determined by the workload parallelism. The benchmarks of the third type also show high scalability, because they have high workload parallelism. The average speedup is lower than that of the first type, because their workload parallelism is lower (Figure 4.1). The benchmarks of the second type and the benchmarks of the fourth type show similar scalability. The average speedups are similar (6.5 and 6.3), despite the differences in synchronization frequency. The average speedups are lower than those of the first and the third types.

Speedups are determined by both workload parallelism and synchronization frequencies on the VM. Though benchmarks with higher workload parallelism still achieve higher speedups than those with lower workload parallelism (e.g., the benchmarks of the first type show higher scalability than those of the fourth type), synchronization frequencies tend to have a larger impact on scalability than workload parallelism. This is evidenced by the benchmarks of the fourth type achieving higher speedups (9.7 on average) than those of the third type (5 on average), though the benchmarks of the fourth type have lower workload parallelism. Synchronizations also make the benchmarks show lower scalability on the VM than on the PM. The average speedups of the benchmarks of the second and the third types are 6.5 and 8.7 on the PM, respectively, and are 3 and 5 on the VM, respectively.

Interestingly, the benchmarks of the last type achieve better scalability on the VM than on the PM. The average speedups are 9.7 on the VM and 6.3 on the PM. This confirms that virtualization inherently improves scalability when synchronizations are infrequent. To better understand how virtualization improves scalability for these benchmarks. We collected the resource utilization efficiencies during their executions on the PM and the VM. Then, for each benchmark, we calculate the ratio

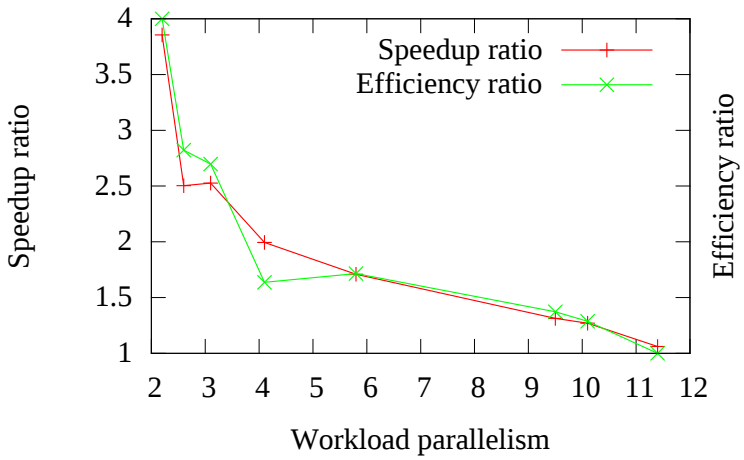

Figure 4.3 Impact of virtualization on scalability for applications with different workload parallelism.

between its speedup on the VM and its speedup on the PM, and the ratio between the efficiencies. Figure 4.3 shows that, for the benchmarks with different workload parallelism, their efficiency ratios are always greater than 1, and the speedup ratios change consistently with the efficiency ratios. This indicates that the scalability improvements are through making more efficient utilization of resources. Figure 4.3 also shows that the speedup ratios decrease with the growth of workload parallelism. This is because, with the growth of workload parallelism, the space for increasing scalability decreases.

4.4 Improving Scalability at the System Level

At the system level, the management of CPU resource has great impact on application scalability on VMs. For best scalability, the system should allocate CPU time in a way that each VM can maximize the utilization of its timeslice. In this section, we first show that there is still much space for improving the existing system design to achieve better scalability. Then, we identify two system factors that can be leveraged to improve scalability.

4.4.1 Potential for Improving Scalability on VMs

To understand the potential for achieving better scalability with improved system designs, we estimated the resource utilization efficiencies that the benchmarks could possibly achieve if the VMM could support the VM to utilize its timeslice as fully as possible. We selected the benchmarks of the fourth type, because 1) we want to focus on improving the allocation of CPU time, instead of reducing the overhead of VCPU switches, and 2) there is space to further improve their scalability.

The estimation is based on profiling the benchmarks on the physical server. For each allocation period during the execution of a benchmark, we collect the CPU time utilization of the benchmark. If the utilization u is higher than the portion p of the CPU time that a VM can obtain (e.g., a 60% utilization vs. 25% of CPU time that a VM can obtain when it is co-located with another 3 VMs), we expect that, with a well-designed VMM, the VM can deplete the timeslice allocated to it and achieve an efficiency of 100% when the computation is executed on the VM. Otherwise, the benchmark does not have enough computation to deplete the timeslice allocated to the VM. Thus, in the period, the efficiency is the ratio between u (utilization) and p (portion of CPU time allocated to a VM). The estimated resource utilization efficiency is the average efficiency during the execution.

Figure 4.4 shows the actual resource utilization efficiency and the estimated maximal efficiency of the benchmarks with a concurrency level of 16. On average, the actual resource utilization efficiency is 62.1% (average speedup is 9.7), and the estimated maximal efficiency is 87.5% (corresponding to a speedup of 16*87.5%=14). The benchmark *p.canneal* shows the largest potential (from 43% to 100%). This clearly shows that there is still much space to further improve the management of CPU resource to achieve higher scalability.

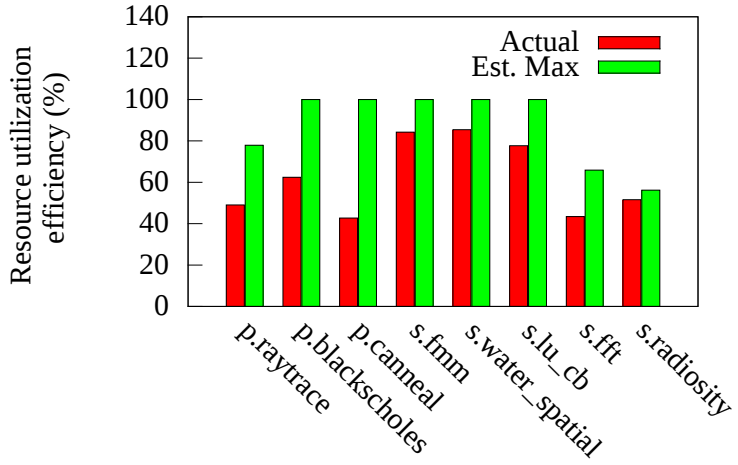

Figure 4.4 Actual resource utilization efficiency and estimated maximal resource utilization efficiency of the benchmarks of the fourth type on a VM.

4.4.2 Possible Optimizations on CPU Time Allocation

Based on the analysis in Section 4.2, application scalability on a VM can be improved by reducing the overhead incurred by VCPU switches and reducing unused timeslice. Extensive research has been conducted on the techniques reducing the impact of VCPU switches on application performance and scalability, such as improving VCPU scheduling at the VMM level and improving task scheduling at the guest OS level [66, 21, 6, 22, 15, 26], or reducing scheduling latencies [53, 67, 68]. Thus, we focus on investigating the factors in CPU time allocation to reduce unused timeslice.

• ***Longer allocation periods***: A VCPU is allowed to use its timeslice within each allocation period. If a VCPU has light workload in an allocation period, it may not deplete its timeslice by the end of the period. The unused timeslice may not be rolled over for the VCPU to handle possibly heavy workload later. Increasing allocation period length can tolerate such workload fluctuation on VCPUs, and reduce unused timeslice of the VCPUs when they have light workload.

To verify the impact of allocation period length on scalability, we repeated the experiments described in section 4.3 for different allocation period lengths from

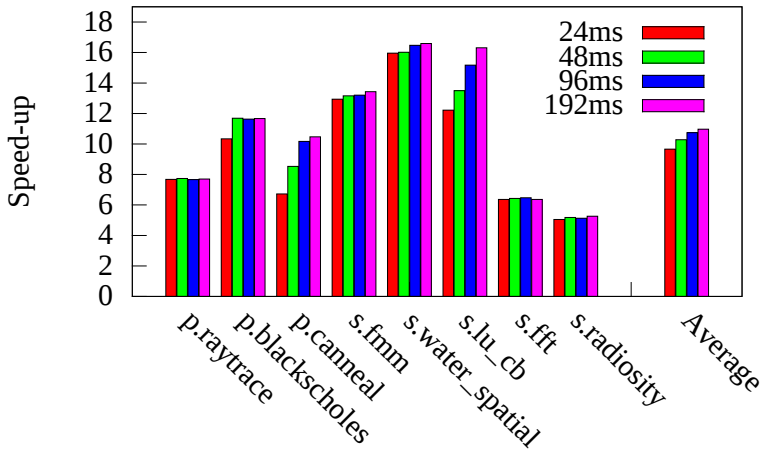

Figure 4.5 Speedups when allocation period length is varied from 24ms to 192ms.

24ms (i.e., system default value) to 192ms, and show the speedups of benchmarks of the fourth type in Figure 4.5. Increasing allocation period length does significantly improve execution scalability for *p.canneal* and *s.lu_cb* and slightly improve execution scalability for *s.radiosity*, *p.blackscholes*, *s.fmm*, and *s.water_spatial*. We also notice that *s.lu_cb* even achieves linear scalability when allocation period length is increased to 192ms. Increasing allocation period length has different impact for different benchmarks because the workloads may fluctuate on different time scales and with different intensity. Though increasing allocation period length cannot increase scalability for *s.fft* and *p.raytrace*, the average speedup of these benchmarks is increased from 9.6 to 11.0 when the allocation period length is increased to 192ms.

These results confirm that longer allocation periods can really improve the execution scalability of multicore applications on VMs. However, increased allocation periods lengthen responding latencies, which may not be desirable for interactive workloads. Thus, long allocation periods may only be applicable to throughput-oriented workloads.

•***Matching resource allocation and workload distribution***: In a VM, workload is distributed to VCPUs for concurrent execution, utilizing the CPU resource (i.e., timeslice) allocated to the VCPUs. Desirable performance can only be achieved when the allocation of timeslice matches the distribution of workload. Allocating more timeslice to a VCPU than what is needed by the workload on the VCPU will cause some timeslice unused by the end of allocation periods. Allocating insufficient timeslice to a VCPU with heavy workload delays the computing tasks on the VCPU.

Matching workload distribution and CPU resource allocation can be done by task scheduling either in guest OSs or applications. Existing VMM designs try to allocate timeslice evenly to VCPUs within each VM. As we will show with an illustrative example in Figure 4.6, task schedulers can evenly distribute workload on the time scale of allocation periods to improve application scalability.

In the example, a program executes a loop, in which each iteration has 2 units of sequential tasks followed by 4 units of parallel tasks. If the program runs on a 4-core PM, the speed-up is 2 based on Amdahl's law. Figure 4.6 shows the executions of the program on a 4-VCPU SMP VM co-located with another VM, and compares the executions with different methods of distributing workloads to VCPUs. We assume that the length of an allocation period is 12 time units and each time unit can finish one unit of task. Two VMs have the same weight. Thus, in an allocation period, each VM is assigned with 50% of CPU time (i.e., 12*4/2=24 units of CPU time), and each VCPU receives 6 units of CPU time.

Subfigure (A) shows the execution of the program on one VCPU. In an allocation period, 6 units of tasks are finished since the VCPU has 6 units of CPU time. Subfigure (B) shows the execution on four VCPUs with a conventional task scheduler. Though parallel tasks can be evenly distributed to the VCPUs, sequential tasks cannot, and are assigned to the same VCPU (VCPU0 in the figure). Thus, only

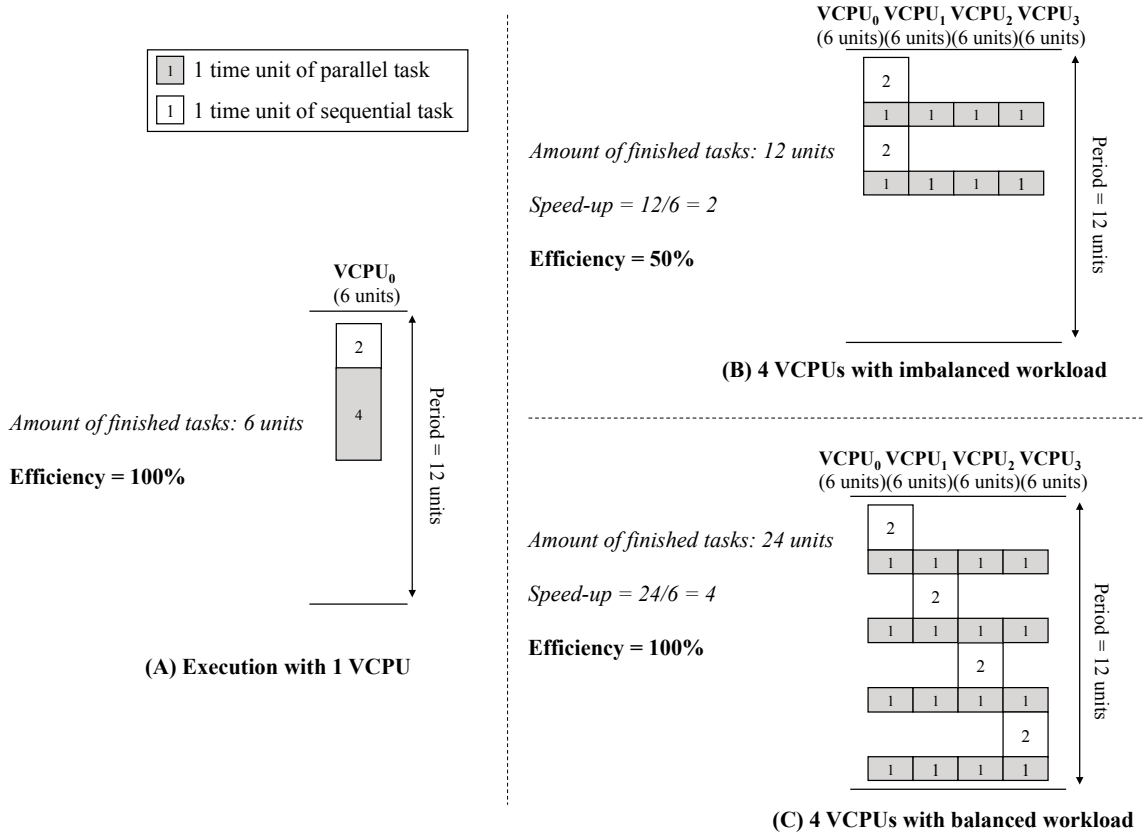

Figure 4.6 An illustrative example to explain the benefit of evenly distributing workload and how even workload distribution can be achieved.

two iterations can be finished within an allocation period. After the second iteration, VCPU0 consumes up its 6 units of CPU time, though other VCPUs only consume 2 units of CPU time each and still have unused timeslice. These VCPUs finish only 12 units of tasks in total in the allocation period, which are 2 times as many as those finished with one VCPU. Thus, the speedup is 2, the same as that on the PM.

Subfigure (C) shows the execution on four VCPUs with an improved task scheduler assigning sequential tasks onto different VCPUs in different iterations. Though the workload is not evenly distributed at every moment, the workload on the VCPUs is balanced on the time scale of allocation periods. With such workload distribution, every VCPU can consume its 6 units of CPU time and finish 6 units

of tasks (four iterations) in an allocation period. With the improved task scheduler, linear speed-up can be achieved, i.e., a speed-up of 4 with 4 VCPUs.

Matching workload distribution and CPU resource allocation can also be done by adjusting the CPU time allocated to VCPUs based on the workload on them. The benefits can be illustrated with the execution shown in Figure 4.6(B). If the VMM can allocate 12 units of CPU time to VCPU0 and 4 units of CPU time to each of the other three VCPUs in each allocation period, the program can still finish four iterations in the period, achieving linear scalability (the figure only shows the first two iterations). Note that the amount of CPU time received by the VM is not increased. The increased scalability comes from distributing more CPU time to VCPU0, which has heavier workload than other VCPUs.

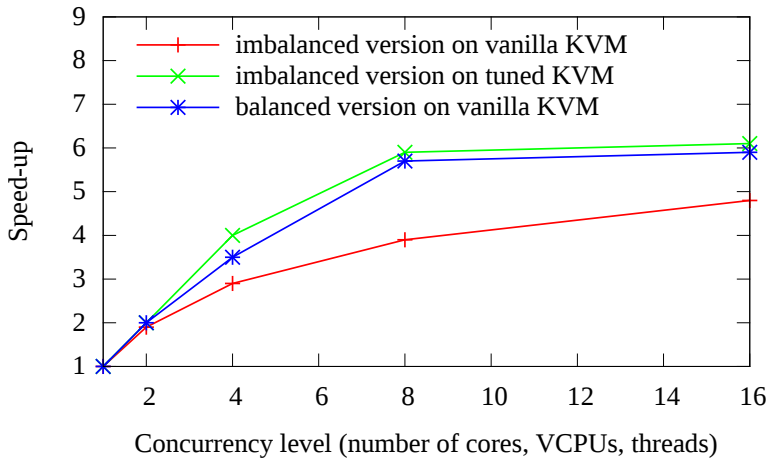

Figure 4.7 Speedups of the synthetic benchmark.

We tested the above approaches using a synthetic benchmark, which generates the workload of typical fork-join multicore programs. Specifically, the benchmark executes a loop, in which each iteration finishes eight units of computation that can be fully parallelized and one unit of computation that can only be executed sequentially. Each unit of computation takes about 1ms to finish. We developed two versions of the synthetic benchmark. In the *imbalanced* version, sequential tasks are executed by

the same thread, as that shown in Figure 4.6(B). In the *balanced* version, sequential tasks in different iterations are assigned to threads in a round-robin manner. This is to emulate the execution with a task scheduler trying to balance the workload on VCPUs.

We first run both versions on a VM managed by the vanilla KVM, which tries to allocate CPU time evenly to VCPUs, and compare their performance. The VM is co-located with three other VMs, each of which runs an instance of the CPU-bound program incrementing a counter. Then, specifically for the imbalanced version, we tuned KVM settings to allocating CPU time to VCPUs proportionally to the workload on them, and rerun the imbalanced version on the VM.

Figure 4.7 shows the speedups of the benchmark when the concurrency level is increased from 1 to 16. The balanced version on vanilla KVM and the imbalanced version on tuned KVM show higher scalability than the imbalanced version on vanilla KVM. When the concurrency level is 16, the speedups are 5.9, 6.1, and 4.8, respectively. This shows that both approaches to match resource allocation and workload distribution are effective to improve scalability. The benchmark cannot achieve linear scalability mainly because there are frequent synchronizations incurred at the beginning and the end of the sequential computation in each iteration.

4.5 Related Work

To understand scalability, analytical models have been developed based on various workload characteristics, such as synchronization and communication [69], critical path [70], memory accessing traffic [71], and the amount of sequential computation. These models target physical machines with dedicated hardware resource, and cannot be directly applied to understand the execution scalability on virtual machines. Various models have been developed in computer architecture area to study how to distribute hardware resource, such as transistors and chip area, to the functional

units in multicore processors to maximize application performance and scalability [72, 73, 74, 75]. They are remotely related with our work.

Targeting application performance on VMs, existing work mainly focuses on characterizing the interference caused by the contention of the shared hardware resource on memory hierarchy (e.g., processor cache and memory bandwidth) between co-located VMs [76, 77, 78, 79, 80, 81]. The main purpose is to understand and alleviate the performance degradation incurred by the interference. This chapter is to identify the application features and system factors affecting the execution scalability of applications on VMs. Existing research on application performance on VMs is orthogonal to our research.

To improve the execution scalability of multicore applications on VMs, various techniques have been attempted at all the system layers, from hardware support (e.g., PLE) [16, 25, 82], VMM [66, 21, 6, 22], guest OSs [26, 24, 57], to programming framework [83]. These techniques only target the virtualization overhead on communication/synchronizations between application threads. This chapter discusses application scalability on VMs in a wider scope, with communication/synchronization being one of the scalability factors.

4.6 Conclusion

This chapter aims to understand how virtualization and CPU resource sharing affect the execution scalability of multicore applications. It does not mean to exhaustively investigate all the factors affecting application scalability (e.g., memory latency, I/O operations). It focuses only on the factors related to CPU time, which is the most important resource for achieving high performance. With analysis and experiments, this chapter shows that application scalability on VMs is mainly affected by a few application features, including workload parallelism and synchronization frequencies, and a few system factors in CPU resource management, including allocation period

lengths and the matching between workload distribution and CPU resource allocation. We hope the findings of this chapter can help cloud computing users gain better understand on the performance of their programs in the cloud and make better choices between physical machines and different types of VM instances. We also hope our work help motivating system researchers and developers to consider the factors limiting scalability and explore practical solutions ameliorating the impact of these factors.

CHAPTER 5

DYNAMICALLY ADJUSTING VIRTUAL CPU FEATURES TO AVOID LOW AND UNSTABLE PERFORMANCE IN BIG VMS

5.1 Introduction

In accordance with the predominance of multicore processors, virtual machines (VMs) are built to have multiple virtual CPUs (VCPUs) for programs to leverage the aggregated computing power of multicore processors. For example, most virtual instances in Amazon cloud are now with 2 or more VCPUs, and the largest instances can have as many as 128 VCPUs. However, on such VMs, multicore programs may suffer significant performance degradation, and their performance can vary widely in an unpredictable way. This problem leads to frustrating user experience in the cloud.

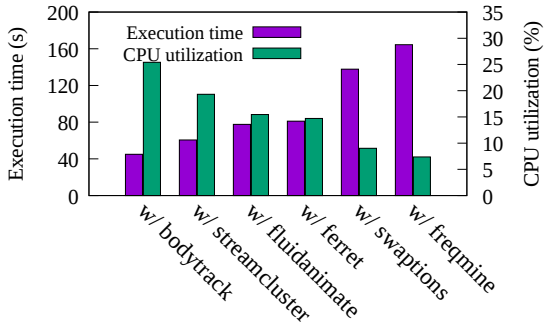

Figure 5.1 Performance and CPU utilization of *bodytrack* when it is executed on a VM colocated with another VM running different benchmarks.

To illustrate this problem, Figure 5.1 shows the performance of *bodytrack*, a program from PARSEC benchmark suite [12], when it is executed on a VM colocated with another VM running different PARSEC benchmarks as labeled on x axis. Each VM has 16 VCPUs. Detailed experiment settings can be found in Section 5.5. Compared to the performance when the two VMs run two instances of *bodytrack* (the first bar in the figure), the performance of *bodytrack* is degraded by 35%∼265% when the other VM runs other benchmarks.

The performance degradation is largely caused by low CPU utilization, and performance variation is caused by the variation of CPU utilization. The CPU utilization in a time period is defined as the ratio between the CPU time utilized by the VCPUs of a VM in the period and the total CPU time on the computing cores of the server. For example, for two colocated VMs, each should get a fair share (50%) of the CPU time of the cores. For a VM, in a time period, the higher its CPU utilization is, the more CPU time it consumes, and the more work it can finish. Thus, high CPU utilization usually leads to high processing speed.

Figure 5.1 shows the CPU utilization of *bodytrack* during its execution and demonstrates a strong corelation between execution time and CPU utilization. For example, when *bodytrack* is colocated with *freqmine*, its execution time is increased to 164s since its CPU utilization is only 7%. The performance degradation and variation can be avoided if the CPU utilization can be fixed to 50%.

Currently, the VCPUs in a VM are built with similar performance features and scheduled with uniform policies, aiming to have similar computing capability. Such a VM is named an SMP-VM or a virtual SMP [22]. For instance, an instance with the largest size in Amazon EC2 has 128 VCPUs, each of which has a computing capacity of 349 EC2 computing units (ECU) [84]. Though the actual computing capability of the VCPUs in a VM may vary, the variation is determined by the colocated workloads contending for CPU time, rather than the demand of the computation on the VCPUs.

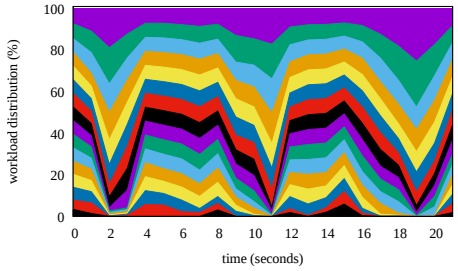

Figure 5.2 Workload distribution of *bodytrack*.

The chapter argues that this symmetric architecture of VMs in term of VCPU management and scheduling is the major obstacle for achieving high and predictable CPU utilization (and thus high and predictable performance). With this design, CPU utilization is low for two reasons. *First*, the symmetric design of VCPUs is achieved by the virtual machine monitor (VMM) trying to evenly distribute CPU time of a VM to its VCPUs. However, low CPU utilization is caused if the workload on the VCPUs is not evenly distributed. When this mismatch between CPU time distribution and workload distribution happens, low CPU utilization is caused on the VCPUs with light workload, since there is not enough computation to utilize much CPU time. At the same time, for the VCPUs with heavy workload, the CPU time available to them cannot be increased to improve the overall CPU utilization of the VM.

As an instance to show that the workload on VCPUs is not evenly distributed, we profiled the execution of *bodytrack* on a 16-VCPU VM running on a dedicated server, and show the workload distribution during the first 20 seconds of the execution in Figure 5.2. In the figure, each color band represents a VCPU, and its width represents the workload on the VCPU (wider bands showing heavier workload). An extreme case for unevenly distributed workload is the sequential portion in a parallel program, during the execution of which only one VCPU has workload and other VCPUs are idle, leading to the lowest CPU utilization.

Second, with the symmetric design, VCPUs in a VM are equally treated by the scheduler and are scheduled with similar priorities. Thus, the VCPUs executing the tasks on critical paths may not be scheduled as early as possible when the tasks are ready. If the tasks on critical paths are delayed, other VCPUs may spend more time on waiting for these tasks to finish. This further reduces the CPU utilization.

Performance variation is caused, because the actual CPU time obtained by each VCPU and how quickly a VCPU can be scheduled are also determined by other VCPUs colocated with the VCPU on the same core, particularly the workload on

these VCPUs. This makes the execution in a VM show different performance when colocated with different workloads. For example, when a VCPU with heavy workload is colocated with another VCPU with light workload (an idle VCPU being an extreme case), the VCPU may get extra CPU time, because of the low CPU utilization of the other VCPU; when a VCPU is colocated with another VCPU that yields the core frequently due to idleness, the VCPU may get scheduled quickly, taking the opportunities of the other VCPU giving up the core. The symmetric design of VCPUs excludes the need to examine the VCPUs to be scheduled on the same core and group the VCPUs carefully to avoid low CPU utilization.

Based on the above observations, this chapter proposes **dynamic asymmetric virtual CPUs**, which have asymmetric performance features and computing capabilities. The asymmetry is achieved by manipulating the amounts of CPU time distributed to VCPUs and the scheduling priorities of the VCPUs based on the demand for CPU time of the computation on the VCPUs. Specifically, more CPU time is distributed to the VCPUs with heavy workload than the VCPUs with light workload; the VCPUs with urgent tasks to finish are scheduled with smaller delays than other VCPUs. Since the demand of the VCPUs may change over time (refer to Figure 5.2), the asymmetry is dynamically adjusted.

Dynamic asymmetric VCPUs help improving the CPU utilization of multicore programs in VMs in three ways. First, it improves the CPU utilization of the VCPUs with heavy workload by assigning them with more CPU time. Second, it reduces the time that VCPUs spend on waiting. Third, the VCPUs scheduled on the same core are examined, and the schedules leading to low CPU utilization are avoided. With improved CPU utilization, serious performance degradation can be avoided, and performance becomes more predictable.

This chapter makes the following contributions. First, to our best knowledge, this is the first work proposing and studying VCPUs with asymmetric performance

features for the desired performance of multicore programs on VMs. Second, this chapter provides a system solution and a set of techniques for building and scheduling asymmetric VCPUs. Third, extensive performance evaluation with PARSEC benchmarks and TPC-W-like workloads shows that the solution can significantly improve the performance and performance stability of multicore programs on virtualized platforms, and can also improve overall system throughput.

The rest of this chapter is organized as follows. Section 5.2 first discusses the related work. Then, the key ideas of creating asymmetric VCPUs are introduced in Section 5.3. Section 5.4 describes the design of the system solution by introducing three key components respectively. We then present the implementation and evaluation of the asymmetric VCPUs. Finally, we conclude our work in section 5.6.

5.2 Related Work

Performance degradation may be caused by various virtualization overhead, e.g., traps into the VMM, memory address translation, high communication/synchronization cost, and overhead associated with increased software layers. To reduce virtualization overhead and the caused performance degradation, various techniques have been attempted at all the system layers, from hardware [16, 25, 82], VMM [66, 21, 6, 22], guest operating systems [26, 24, 57], to programming framework [83]. These techniques for reducing virtualization overhead are orthogonal to the proposed solution and can work together synergistically.

Performance degradation and performance unpredictability due to the interference between colocated VMs have been widely noticed and analyzed [27, 76, 77, 85, 79, 80, 86, 87]. Most of the works focus on the contention for shared resources on memory hierarchy, e.g., shared cache space, memory bandwidth, storage bandwidth. There are proposals to enhance hardware and VMM designs to include the management of these resources [88, 89, 90, 91, 92, 93, 94, 95, 96]. They are

orthogonal to the research in this chapter. There are also solutions on selecting colocating VMs [85, 97, 98, 87, 99, 79]. They target single-VCPU VMs, and cannot be applied to the VMs with multiple VCPUs running multicore workloads.

Targeting colocated parallel applications, Callisto provides a runtime system to reduce the performance interference between applications through reduced resource sharing and synchronization overhead [100]. It requires that applications be developed based on Callisto. Our solution is general and can be applied to any multicore applications. A delayed preemption mechanism was proposed in [86] to reduce the interference caused by the VCPUs in one VM frequently preempting the VCPUs in another VM. However, the management of VCPUs are still symmetric.

Asymmetric systems are built to accelerate performance bottleneck and have become a well-recognized approach to achieving high performance. On physical machine, the techniques to achieve asymmetric performance on multicore processors fall into two categories. First, asymmetric multicore processors (AMP) are designed to have cores with the same instruction set architecture but different performance. Compared to symmetric multicore processors, AMP show significant advantages in performance and energy efficiency [72, 101]. Second, asymmetric performance can be achieved on a symmetric multicore processor with techniques, such as DVFS and turbo-boost, which dynamically increase the frequencies of specific cores to boost their performance [102, 103]. Our solution follows the same direction, but applies to virtual machines. On virtualized platforms, since CPU cores are shared by multiple VMs and CPU time allocation is controlled by software, it is more effective and more flexible to create and deliver asymmetric performance desired by applications than on physical machines.

The virtual asymmetric multiprocessor design [104] shares some similar ideas as our solution. It targets virtual desktop workloads and aims to improve user experience by reducing the latency of interactive tasks. To achieve this objective, it detects the

VCPUs running interactive tasks in each VM, and allocates more CPU time to these VCPUs than other VCPUs in the VM. The solution aims to improve the performance of interactive tasks. Our solution targets the overall throughput of a VM, and aims to avoid performance degradation and unpredictable performance by making best utilization of the CPU time available to the VM.

5.3 Key Ideas and General Approach

The objective of designing dynamic asymmetric VCPUs is to distribute the CPU time of a VM to its VCPUs in a way that the CPU time can be fully utilized by the computation on the VCPUs, so that the computation can proceed at the highest speed[1]. The basic idea is to dynamically adjust the allocation of CPU time to VCPUs, such that the allocation matches the demand of the computation tasks running on the VCPUs[2]. Since CPU time is allocated to VCPUs periodically, the adjustment is also made periodically. Specifically, at the beginning of each allocation period, the VMM predicts the CPU time demand on each VCPU in this period and controls the CPU time allocation accordingly.

Implementing the idea needs to address two key issues. One is how to characterize the CPU time demand on VCPUs. The other is how to predict the CPU time demand on VCPUs at the VMM level.

[1]We assume that more work can be finished with more CPU time and higher CPU utilization leads to better performance, since on virtualized platforms busy-waiting on VCPUs is monitored by hardware and minimized [105].

[2]Please note that, in our solution, the CPU time demand on each VCPU is not the demand for the CPU time that can be used to complete all the unfinished tasks on the VCPU, or the demand for the CPU time that can be used by the VCPU to complete the largest possible amount of work itself locally in the forth-coming period. Instead, the CPU time demand on each VCPU refers to the demand, by satisfying which the CPU utilization of the VM can be maximized and the workload running in the VM can finish the largest possible amount of work in the forth-coming period.

5.3.1 Amount and Urgency of CPU Time Demand

A multi-threaded workload running on a VM can be considered as a set of tasks distributed on the VCPUs of the VM. There are dependencies between tasks. A task must wait for its dependencies to be satisfied before it can proceed. Thus, the best performance relies on the tasks on all the VCPUs in the VM to make progress concurrently and in a coordinated way (i.e., minimizing one VCPU waiting for another VCPU). Thus, CPU time must be allocated to support such execution.

We use two features to characterize CPU time demand. *One feature is amount, i.e., how much CPU time is needed by the workload on each VCPU in a period to achieve the best performance (i.e., highest CPU utilization).* The amount of CPU time needed by a VCPU depends on the amount of work the VCPU can finish if the VCPUs in the VM make progress concurrently and in an coordinated way, which is in turn determined by the workload distributed on the VCPU.

Different VCPUs may need different amount of CPU time because applications may not be able to distribute workload evenly to VCPUs. More work needs more CPU time to finish. For example, for the applications modeled in Amdahl's law, parallel computation is distributed evenly to all the VCPUs and sequential computation is done by one of the VCPUs. Thus, the VCPU executing the sequential part receives more work than other VCPUs. Some applications follow the pipeline model, in which different pipeline stages may have different processing time. In these applications, different VCPUs handle different stages and finish different amount of work.

The other feature is urgency, which describes how quickly the CPU time should be made available when a VCPU becomes ready to run. A VCPU can use its CPU time only when it is scheduled. On a virtualized platform, CPUs are time-shared by multiple VMs. Thus, a VCPU may not be scheduled immediately when its task is ready to run. For example, when all the cores are being used, a VCPU must wait for another VCPU being descheduled before it is scheduled. This delays the task on the

VCPU. We refer to the delay between the time when a VCPU becomes ready to run and the time when CPU time is made available to the VCPU as *VCPU scheduling latency*. VCPU scheduling latency may reduce the amount of work that VCPUs can finish in a period, and lower CPU utilization.

The delay caused by VCPU scheduling latency can accumulate. For example, for a chain of tasks with dependencies between them, the delay of scheduling one task will postpone the scheduling of all the subsequent tasks; the last task will see all the delays of its antecedent tasks.

Different VCPUs may tolerate different levels of latency. For example, if the task on a VCPU is on critical path, the VCPU requires a low VCPU scheduling latency to reduce the delay of subsequent tasks. If the task on a VCPU has much work to finish, the VCPU may need a low VCPU scheduling latency to prevent the task from becoming a task on critical path. If the task on a VCPU has only a small amount of work or the task is not on the critical path, the VCPU may tolerate a relatively high VCPU scheduling latency, i.e., the CPU time demand is of low urgency.

5.3.2 Predicting CPU Time Demand

Due to the semantic gap between applications and the VMM, at the VMM level, it is challenging to obtain the CPU time demand information of applications. The VMM only has the information on whether VCPUs are ready to run, which is used to schedule ready VCPUs and deschedule idle VCPUs. (A VCPU becomes idle when the dependencies of its task are not satisfied.) However, the information can hardly be used for understanding the CPU time demand of computation tasks.

To address this issue, the VMM predicts the CPU time demand of VCPUs for the forthcoming period based on how they utilize the CPU time allocated to them in the past period, and adjusts the prediction in a direction that can make the VM achieve a better utilization of its CPU time in the forthcoming period.

For example, a VCPU is provided with an amount of CPU time in a period. If the VCPU fails to fully utilize the CPU time at the end of the period, the fact indicates that the CPU time demand of the VCPU is not that high. Thus, the CPU time that the VCPU consumes in the current period is used to predict the amount of CPU time it demands in the forthcoming period. This, on one hand, prevents the VCPU from wasting CPU time in the forthcoming period, and, on the other hand, makes more CPU time available to the VCPUs where CPU time can be better utilized. If the VCPU has fully utilized the CPU time at the end of the current period, the fact indicates that the CPU time demand of the VCPU is higher than what has been predicted, and it should be allocated with more CPU time in the forthcoming period to finish more work.

5.4 System Solution

This section introduces an integrated system solution for building dynamic asymmetric virtual CPUs. The solution includes three key components: (1) a CPU time allocation component component for predicting the amount of CPU time demanded by the workload on VCPUs and allocating CPU time accordingly, (2) a scheduling latency adjustment component for predicting the urgency level of the demand on each VCPU and adjusting scheduling latency accordingly, and (3) a resource conflict resolver for detecting conflicting demand of VCPUs and resolving conflicts by migrating VCPUs between cores. For the first two components, we focus on introducing how they predict CPU time demand, since satisfying the CPU time demand is just to control the allocation of CPU time and VCPU scheduling based on the prediction, and is system-dependent.

5.4.1 CPU Time Allocation Component

The CPU time allocation component collects the amount of CPU time consumed by each VCPU periodically and uses the amount of CPU time consumed by the VCPUs in the previous period to adjust the prediction of the CPU time demand for the upcoming period. Specifically, if a VCPU has been allocated with sufficient CPU time in a period, the CPU time actually consumed by a VCPU indicates the amount of work it finishes in the period. Thus, it is used to predict its CPU time demand in the upcoming period. However, if a VCPU has been allocated with insufficient CPU time in the period, the amount of work it finished in the period is limited by the CPU time available to it, and cannot reflect its actual demand for CPU time. Since it is not possible to accurately predict its actual demand, we gradually increase the prediction until the VCPU is allocated with enough CPU time after a few periods. In our implementation, we increase the prediction by 10% every time if a VCPU cannot consume its CPU time.

To check whether a VCPU has been allocated with sufficient CPU time, the CPU time allocation component looks at the status of the VCPU. If the VCPU is in a "ready" state (i.e., it can still make progress), it is safe to assume that the demand of the VCPU has not been satisfied and the VCPU needs more CPU time in the next period.

It is possible that the aggregated CPU time demand predicted by the CPU time allocation component is not equal to the total CPU time available to the VM. Thus, when the predicted CPU time demand values have been determined, we use those values as weights, and distribute the CPU time of the VM to the VCPUs based on the weights. Then, we change the predicted VCPU time of each VCPU based on the amount of CPU time distributed to it. For example, in a 2-VCPU VM, VCPU0 was assigned with 40ms CPU time, but consumed 35ms in a period; VCPU1 was assigned with and actually consumed 40ms CPU time in the period; the predicted

CPU demand values of two VCPUs are 35ms and 44ms (i.e., 40×1.1), respectively. If the VM gets 80ms CPU time in each period, VCPU0 will get 35.4ms CPU time (i.e., 80ms×35/(35+44)) and VCPU1 will get 44.6ms CPU time (i.e., 80ms×44/(35+44)). Their predicted CPU demands are changed to 35.4ms and 44.6ms, accordingly.

5.4.2 Scheduling Latency Adjustment Component

To determine whether a VCPU has urgent CPU time demand, at the end each period, the scheduling latency adjustment component looks at whether the VCPU has consumed its allocated CPU time and whether the VCPU can still make progress. If the VCPU has not consumed its allocated CPU time and the VCPU can still make progress, it is possible that the CPU time is made available too late to the VCPU during the period. Thus, the scheduling latency adjustment component determines that the VCPU have more urgent CPU time demand than that predicted previously, and decreases the scheduling latency of the VCPU[3]. If the VCPU has consumed its allocated CPU time and the VCPU is idle at the end of the period, the scheduling latency adjustment component determines that the demand of VCPU is not urgent and increases its scheduling latency.

There are scenarios, in which a VCPU with a low scheduling latency has tasks depending on the completion of the tasks on other VCPUs with high scheduling latencies. Since the tasks on the VCPUs with high scheduling latencies complete late, the task on the VCPU with a low scheduling latency cannot start early. Thus, it is possible that the VCPU with a low scheduling latency still cannot consume its CPU time, no matter how its scheduling latency is reduced. To detect such scenarios, when the scheduling latency of a VCPU has been reduced to a minimal value allowed by the system, if a VCPU still cannot consume its CPU time, the

[3]Low scheduling latency increases VCPU switches and the associated overhead. To address this issue, we take the VCPU switch overhead caused by a VM as a penalty to charge the CPU time allocated to the VM.

scheduling latency adjustment component assumes that the VCPU may be delayed by other VCPUs with high scheduling latencies. To pin-point these VCPUs, the scheduling latency adjustment component uses wake-up inter-processor interrupts (IPIs) sent to the VCPU as indicators to find out the source VCPUs sending out the IPIs. Then it reduces the scheduling latencies of these source VCPUs.

Depending on system implementations, adjusting scheduling latency can be implemented using different mechanisms, such as adjusting wake-up latency parameters in KVM and Linux.

5.4.3 Resource Conflict Resolver

The adjustment of CPU time and scheduling latency of the VCPUs based on their CPU time demand effectively make the VCPUs having asymmetric performance features. When scheduling such VCPUs, care must be taken to avoid resource conflicts. Resource conflicts arise when the total amount of CPU time demand of the VCPUs scheduled on the same core exceeds the core's capacity. For example, a conflict arises when, in a time period of 80ms, two VCPUs scheduled on the same core are allocated with 50ms CPU time each. VCPUs with urgent CPU time demand also have conflict. Since a core is time-shared by VCPUs, it may not be able to satisfy the urgent demand from multiple VCPUs simultaneously. A conflict arises when a core is running a VCPU with urgent CPU time demand and another VCPU with urgent CPU time demand becomes ready to run. If the former VCPU is preempted promptly, its task cannot be finished quickly. If the former VCPU is not preempted promptly, the latter VCPU cannot be scheduled quickly.

The resource conflict resolver detects resource conflicts. If there are conflicts detected, it resolves conflicts by adjusting the layout of VCPUs on physical cores. Since adjusting VCPU layout is costly, the resource conflict resolver does the adjustment in a conservative way. Specifically, to detect and resolve conflicts caused

by high demands for CPU time, after the CPU time allocation component has adjusted the CPU time allocated to each VCPU, for each core, the resource conflict resolver calculates an aggregated CPU time of the VCPUs scheduled on the core. Then, the resource conflict resolver finds out the core with the largest aggregated CPU time and the core with the smallest aggregated CPU time. If the largest aggregated CPU time is greater than the smallest aggregated CPU time by 10%, the resource conflict resolver tries to balance the aggregated CPU time by swapping some of the VCPUs on the two cores.

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

5.5 Evaluation

To demonstrate the performance advantages of dynamic asymmetric VCPUs and test our system solution, we have implemented a system software prototype based on KVM and Linux (Section 5.5.1). Then, we run the benchmarks from PARSEC 3.0 suite [12] and DBT-1 from OSDL database test suite [106], on vanilla KVM and our prototype, and compared their performance. The experiments show that dynamic asymmetric VCPUs can significantly improve the performance (Section 5.5.3) and scalability (Section 5.5.6) of multicore applications on VMs, reduce response time (Section 5.5.5, and increase the overall system throughput of the physical server (Section 5.5.4). With dynamic asymmetric VCPUs, the performance of a multicore application becomes more stable/predictable when it is colocated with different applications (Section 5.5.7). We also show how each component in our solution helps improving application performance in Section 5.5.8.

5.5.1 Prototype Implementation

The solution described in Section 5.4 is based on conventional scheduler designs, where CPU time is explicitly assigned to VCPUs periodically and the CPU time used by VCPUs is book-keeped by the scheduler. In KVM, a VCPU is implemented as an entity scheduled by the Complete Fair Scheduler (CFS) in Linux. CFS does not maintain the CPU time used by each VCPU. Instead, it maintains a virtual runtime for each VCPU, and schedules VCPUs based on their virtual runtimes. There is not explicit CPU time allocation. A VCPU can run as long as its virtual runtime is low enough. In our implementation, accounting feature of the cgroup [107] is used to collect the CPU time usage of each VCPU. CPU time allocation is implemented by adjusting the *share* parameter of each VCPU within a VM. To determine whether a VCPU has depleted the CPU time allocated to it, the implementation compares the CPU time allocated to it and the CPU time it has consumed (collected with cgroup).

To adjust VCPU scheduling latencies, the implementation changes the *sched_wakeup_granularity_ns* parameters. With existing KVM/Linux design, a system wide *sched_wakeup_granularity_ns* parameter is used to tune how quickly a VCPU can be scheduled when the task on it become ready to make progress. To enforce different scheduling latencies for different VCPUs, the implementation creates and maintain a private *sched_wakeup_granularity_ns* for each VCPU in cgroup.

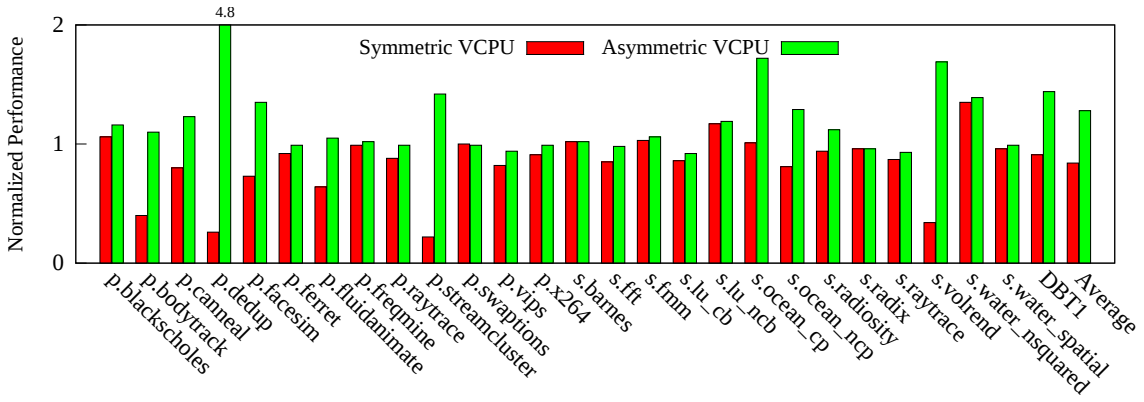

Figure 5.3 Normalized performance with symmetric VCPUs and dynamic asymmetric VCPUs. The number of threads that can run concurrently in each VM is 16.

Please note that the mechanisms used to control CPU time allocated to each VM, including the weights of VMs, are not changed in the prototype. Thus, the performance improvements are mainly from the changes of the way in which CPU time is distributed to VCPUs.

5.5.2 Experimental Setup

We conducted our experiments on a Dell PowerEdge R720 server with 64GB of DRAM and two 2.40GHz Intel Xeon E5-2665 processors. Each processor has 8 cores. On the server, we created 4 VMs with 16 VCPUs. Each VM has 16GB of memory. The VMM is KVM. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. The VCPUs in each VM were laid out on the cores in a way to prevent VCPU stacking for better performance [22]. Low power modes

of cores can reduce the performance of the applications running in VMs [11]. To prevent such performance degradation, in the experiments, we disabled the C states other than C0 and C1 of the processors, which have long switching latencies.

We used the DBT-1 in OSDL database test suite. It simulates the activities of web users who browse and order items from an on-line bookstore. It generates a database workload with the same characteristics as that in the TPC-W benchmark specification [108]. The database generated for the experiments includes information on 100,000 items and 2.9 million customers.

We also used the benchmarks in PARSEC 3.0 suite, including native PARSEC benchmarks and SPLASH2X benchmarks in the suite. We excluded SPLASH2X's cholesky benchmark since its execution time was too short on our system to be reliably measured. We attach a prefix 'p.' before the name of each native PARSEC benchmark, and attach a prefix 's.' before the name of each SPLASH2X benchmark, in order to differentiate these two sets of benchmarks. We also refer to native PARSEC benchmarks as PARSEC benchmarks for brevity. In addition to the benchmarks in PARSEC 3.0 suite, we also selected a micro-benchmark named *MatMul*, which is a multi-threaded CPU-bound program multiplying two matrices of 8000×8000 integers.

We compiled the PARSEC and SPLASH2X benchmarks using gcc with the default settings of the gcc-pthreads configuration in PARSEC 3.0. The gcc compiler and the libraries required by the benchmarks are stock software components in the Ubuntu Linux distribution. We used the parsecmgmt tool in the PARSEC package to run the benchmarks with native input. By default, we run each benchmark with 16 threads.

5.5.3 Performance Improvement

In this section, we show that dynamic asymmetric VCPUs can prevent the serious performance degradation of multicore applications by substantially improve

their performance when they are colocated with other workload, particularly with CPU-bound workload. We run DBT-1 and each of the PARSEC and SPLASH2X benchmarks in the following scenarios:

- Four instances of the benchmark are run in the four VMs with symmetric VCPUs, one in each VM. The VMs are managed by vanilla KVM and Linux.
- One instance of the benchmark is run in one of the four VMs with symmetric VCPUs. Three instances of *MatMul* are run in the other three VMs, one on each VM. The VMs are managed by vanilla KVM and Linux.
- One instance of the benchmark is run in one of the four VMs with dynamic asymmetric VCPUs. Three instances of *MatMul* are run in the other three VMs, one on each VM. The VMs are managed by our prototype implementation.

We compare the performance of the benchmark in the latter two scenarios to show the performance advantage of asymmetric VCPUs. Since the absolute execution times vary widely across different benchmarks, we normalize the performance measured in these scenarios against the baseline performance, which is the performance of the benchmark measured in the first scenario. To be consistent, we use large values to represent higher performance.

We select the latter two scenarios because multicore applications usually show the largest performance degradation when they are colocated with CPU-bound applications (refer to the experiments in Section 5.5.7). Thus, the experiments can demonstrate the greatest potential of dynamic asymmetric VCPUs on improving performance and preventing performance degradation. We use the performance in the first scenario as baseline performance, because, with similar workloads on the VMs, the VMs can obtain the same amount of CPU time, i.e., a fair share of CPU time. Thus, the performance represents a "normal" performance that can be achieved.

Figure 5.3 shows the normalized performance of the benchmarks in the latter two scenarios. Since the baseline performance is always 1, it is not shown in the figure. In the figure, the bars lower than 1 indicate performance degradation, compared to the

"normal" performance, and the bars higher than 1 indicate performance improvement. Out of the 27 benchmarks, 17 benchmarks show performance degradation by more than 2% with symmetric VCPUs. The largest performance degradation is with *streamcluster*, whose execution time is lengthened by almost 5x. On average, the performance is dropped by 16% to 0.84 with symmetric VCPUs, compared with the "normal" performance.

The performance degradation can hardly be found with dynamic asymmetric VCPUs. Only two benchmarks show observable performance degradation. The largest performance degradation is 8% with *lu_cb*. Most benchmarks show performance improvements. The largest performance improvement is with *dedup* (4.8x). On average, the performance of these benchmarks is increased by 28% with dynamic asymmetric VCPUs. The performance improvements are observed because multicore applications may not achieve the best performance with symmetric VCPUs even when they can obtain a fair share of CPU time (the first scenario).

Comparing the performance in the latter two scenarios, we find that the benchmarks can always achieve better performance with dynamic asymmetric VCPUs than they do on symmetric VCPUs. On average, the performance is increased by 52% from 0.84 to 1.28. The largest performance improvement is 18.2x with *p.dedup*.

5.5.4 System Throughput Improvement

To show that dynamic asymmetric VCPUs can improve the overall system throughput of the physical server shared by the VMs running multi-threaded workloads, we run DBT-1 and each of the PARSEC and SPLASH2X benchmarks in the following scenarios, and compare the performance. This is motivated by the observation from the experiments in the previous subsection, which shows that multicore applications may not achieve the best performance with symmetric VCPUs even when they can obtain a fair share of CPU time. With dynamic asymmetric VCPUs improving

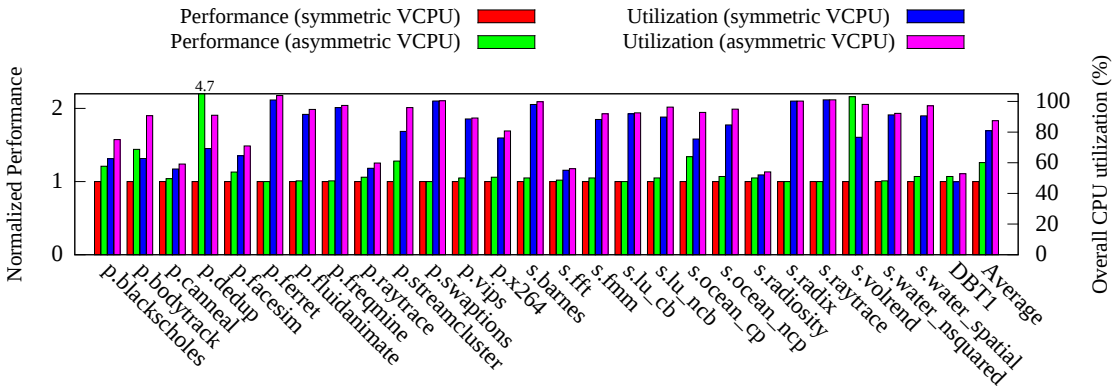

Figure 5.4 Normalized performance and CPU utilization of PARSEC and SPLASH2X benchmarks on symmetric VCPUs and dynamic asymmetric VCPUs. The number of threads that can run concurrently in each VM is 16.

CPU utilization, each of the colocated applications can obtain a larger fair share of CPU time, leading to a higher performance for each application and a higher system throughput for the physical server.

- Four instances of the benchmark are run in the four VMs with symmetric VCPUs, one in each VM. The VMs are managed by vanilla KVM and Linux.
- Four instances of the benchmark are run in the four VMs with dynamic asymmetric VCPUs, one in each VM. The VMs are managed by our prototype implementation.

Since the instances of the same benchmark are run on the colocated VMs, the instances show similar performance. The average performance of the instances is used as the throughput of the physical server. To show that the improved throughput is from increased CPU utilization, we also collected the overall CPU utilization of the VMs. Figure 5.4 shows the normalized performance for all the benchmarks, using the performance in the first scenario as the baseline performance. It also shows the CPU utilization in these two scenarios.

As shown in the figure, all the benchmarks show better or similar performance with dynamic asymmetric VCPUs, compared to that with symmetric VCPUs. Out of 26 benchmarks, 16 benchmarks show observable performance improvements. The

largest performance improvement is with *dedup* (4.7x). On average, the performance is improved by 26% for all the benchmarks with dynamic asymmetric VCPUs. The performance improvements (scenario 2 vs. scenario 1) are similar to those achieved in the experiments in the previous subsection (scenario 3 vs. scenario 1). This indicates that with dynamic asymmetric VCPUs, performance can be improved consistently, regardless of different co-located workloads.

The performance improvements are through improved CPU utilization. As shown in the figure, all the benchmarks make better CPU utilization with dynamic asymmetric VCPUs. On average, the overall CPU utilization is increased by 9% from 81% to 88% with dynamic asymmetric VCPUs. The performance improvement is higher than the increase of CPU utilization. This is because the total amount of computation is reduced for *dedup* and *volrend* when executed on dynamic asymmetric VCPUs, compared to the executions on symmetric VCPUs. This leads to lower CPU utilizations with these benchmarks on dynamic asymmetric VCPUs. For the other 24 benchmarks, the performance is improved by 8% on average, and the CPU utilization is increased by 7% on average. The largest increase in CPU utilization is 45% with *p.bodytrack*. This is translated into a 44% performance improvement.

We also noticed that, even on dynamic asymmetric VCPUs, the overall CPU utilization is not close to 100% for some benchmarks. There are two reasons. First, some benchmarks are not computation-intensive (at least in some execution phases, e.g., long time spent on sequential computation). Second, some virtualization overhead, such as the time spent in VMM, are not counted as the CPU utilization of the applications.

5.5.5 Reducing Response Time

For DBT-1, we also compared the average response time of different types transactions of the last two scenarios in the experiments in Section 5.5.3 and the two scenarios in

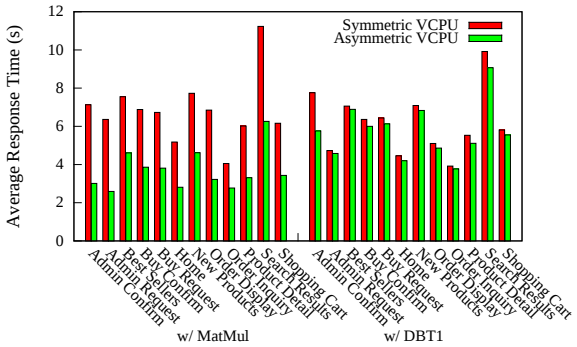

Figure 5.5 Average response time of different types of transactions in DBT-1.

the experiments in Section 5.5.4. As shown Figure 5.5, dynamic asymmetric VCPUs can reduce the response times for all the types transactions across both scenarios. When DBT-1 is colocated with MatMul, the average response time is significantly reduced by 46% from 6.8s to 3.7s. When multiple instances of DBT-1 are colocated, the average response time is slightly reduced from 6s to 5.7s. The solution shows more potential for the scenarios with colocated DBT-1 and MatMul, since the lengthened response time is largely caused by unavailability of CPU time. With dynamic asymmetric VCPUs, DBT-1 shows longer response times when collocated with other DBT-1 instances than it with MatMul, because multiple DBT-1 transactions may compete the same type of hardware resources, such as cache spaces and memory bandwidth.

5.5.6 Scalability Improvements

In the previous two subsections, the benchmarks were executed with a concurrency level of 16. (The number of VCPUs in each VM, the number of cores used to run these VCPUs, and the number of threads in each VM are 16.) To show that dynamic asymmetric VCPUs can improve application performance consistently for different concurrency levels, we conducted the third set of experiments, in which each benchmark was ran in the following scenarios. For each scenario, we vary the concurrency level from 1 to 16.

- One instance of the benchmark is run in one of the four VMs with symmetric VCPUs. Three instances of *MatMul* are run in the other three VMs, one on each VM. The VMs are managed by vanilla KVM and Linux.
- One instance of the benchmark is run in one of the four VMs with dynamic asymmetric VCPUs. Three instances of *MatMul* are run in the other three VMs, one on each VM. The VMs are managed by our prototype implementation.
- Four instances of the benchmark are run in the four VMs with symmetric VCPUs, one in each VM. The VMs are managed by vanilla KVM and Linux.
- Four instances of the benchmark are run in the four VMs with dynamic asymmetric VCPUs, one in each VM. The VMs are managed by our prototype implementation.

Since we want to check whether dynamic asymmetric VCPUs can improve performance for different concurrency levels, we selected six benchmarks, with which dynamic asymmetric VCPUs can effectively improve performance. For each of the benchmarks, we calculated its speed-up by dividing the execution time with the concurrency level equal to 1 by the execution time with the concurrency level larger than 1. We compare the speed-ups for the scenarios in which a multicore application is co-located with CPU-bound workload and the application may not get a fair share of CPU time (the first two scenarios). Then, we compare the speed-ups for the scenarios in which a multicore application can get a fair share of CPU time (the last two scenarios).

The speed-ups of the benchmarks in the first two scenarios are presented in Figure 5.6, and the speed-ups in the last two scenarios are compared in Figure 5.7. Both figures show that applications always achieve higher scalability on dynamic asymmetric VCPUs than they do on symmetric VCPUs. We also observed that the application performance on symmetric VCPUs may start to saturate at lower concurrency levels. For example, as shown in both figures, on symmetric VCPUs, the performance of *s.ocean_cp* and *volrend* starts to drop at low concurrency levels of 4 or 8. However, on dynamic asymmetric VCPUs, their performance scales well when the concurrency level is increased to 16. The largest scalability improvement is with

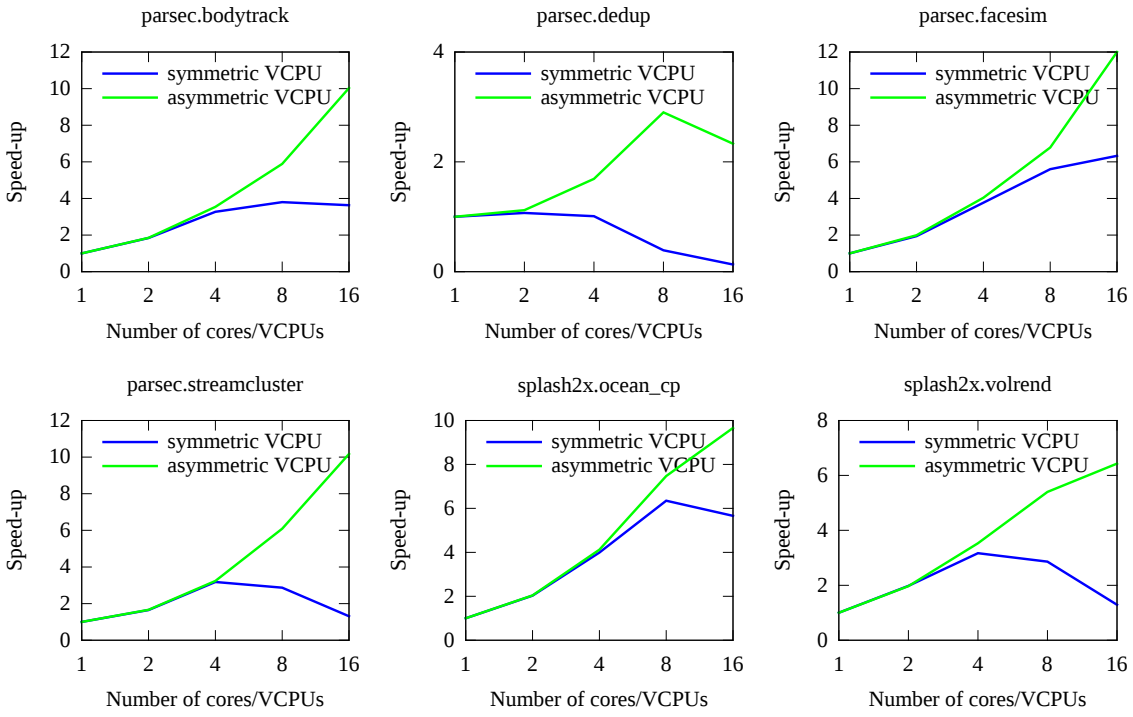

Figure 5.6 Speedups of six benchmarks when concurrency level is varied from 1 to 16 (each benchmark is co-located with CPU-bound workload).

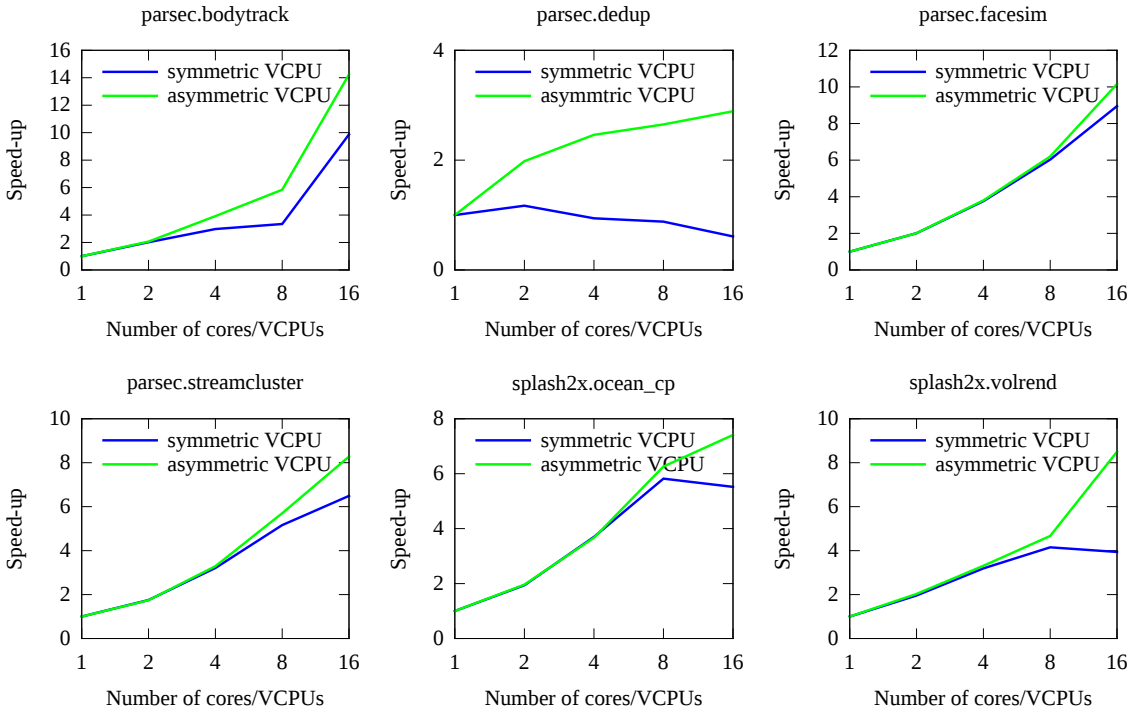

Figure 5.7 Speedups of six benchmarks when concurrency level is varied from 1 to 16 (four instances of the same benchmark are colocated).

p.dedup. At the concurrency level of 16, its performance is improved by 18x and 5x for different scenarios. We also noticed that the benchmarks tend to achieve better scalability when four instances of each benchmark are colocated than they do when co-located with CPU-bound workloads.

5.5.7 Improvements on Performance Stability

To show that dynamic asymmetric VCPUs can improve the performance stability, we conducted the fourth set of experiments in which the selected six benchmarks in previous experiments were ran in the following scenarios:

- Two instances of the benchmark are run in the two VMs with symmetric VCPUs, one in each VM. The VMs are managed by vanilla KVM and Linux.
- One instance of the benchmark is run in one of the two VMs with symmetric VCPUs. Another instance of *MatMul*, *streamcluster*, or *fluidanimate* is run in another VM. The VMs are managed by vanilla KVM and Linux.
- One instance of the benchmark is run in one of the two asymmetric VMs. Another instance of *MatMul*, *streamcluster*, or *fluidanimate* is run in another VM. The VMs are managed by KVM and Linux with our prototype implementation.

Performance variation is usually caused by the resource contention between VMs (i.e., inter-VM interference). We show that the performance of the application on the asymmetric VMs is more stable and resistant to interference than that in the symmetric VM by comparing how the performance varies when the six benchmarks are run in the latter two scenarios when the VM is co-located with another VM running different types of workloads. Targeting CPU resource, we select three co-running benchmarks to generate different level of resource contention. The *MatMul* is computation intensive and provides persistent interference. *Streamcluster* and *fluidanimate* both consist of synchronization and incur intermittent interference. More specifically, *streamcluster* employs the fine-granular synchronization and its computation tasks are evenly distributed over the VCPUs, and *fluidanimate* uses

the coarse-granular synchronization and its computation tasks are spread in an imbalanced way across the VCPUs. The results are shown in the Figure 5.8 and the Figure 5.9. We use the performance in the first scenario as the baseline to normalize the performance.

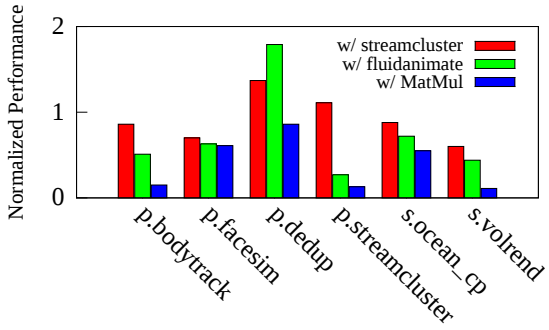

Figure 5.8 Performance variations of the six benchmarks on symmetric VMs under different interferences.

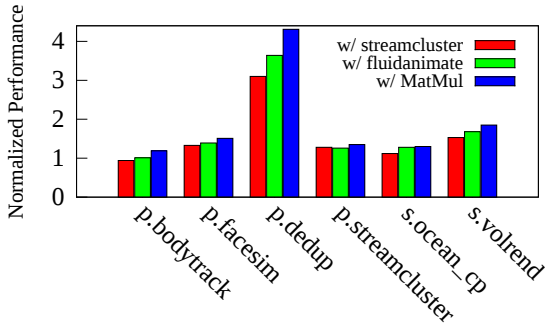

Figure 5.9 Performance variations of the six benchmarks on asymmetric VMs under different interferences.

As shown in the Figure 5.8 and the Figure 5.9, the benchmark performance on the asymmetric VMs is more stable and better as well. On average, the standard deviation of the performance on asymmetric VMs under different interferences is 2.6, which shows much less variation compared to that (4.3) on symmetric VMs. The reason is that dynamic asymmetric VCPUs can maximize the CPU utilization by meeting the resource demand of the computation which makes itself more resistant to the resource contention.

In the Figure 5.8, we observed that most benchmarks perform best when co-running with streamcluster, and perform worst when co-running with *MatMul* program. The reason is that VCPU running urgent and resource-demanding task cannot be quickly served with more CPU resource under more persistent interference. Moreover, fluidanimate generates harmful asymmetric CPU resource contention on cores with its imbalanced workload. For example, a core with higher resource contention could significantly degrade the performance of a VCPU running critical task, which hurts the overall performance of the benchmark. In the Figure 5.9, we found that most benchmarks perform best when co-running with *MatMul* program, and perform worst when co-running with streamcluster. The reason is that when with CPU utilization being optimized on asymmetric VMs, the fine-granular synchronization in streamcluster cause more costly VCPU switches which reduce the overall available CPU resource. Also, imbalanced workload in fluidanimate creates more flexibility and opportunities for the asymmetric VCPUs to reduce the resource contention. For example, a critical task can be moved to a core with less resource contention.

5.5.8 Performance Improvement Breakdown

To show how each component in the proposed system solution helps to improve the performance, we conducted the fifth set of experiments in which the six selected benchmarks were ran in the following scenarios:

- Four instances of the benchmark are run in the four symmetric VMs, one in each VM. The VMs are managed by vanilla KVM and Linux.
- Four instances of the benchmark are run in the four asymmetric VMs, one in each VM. The VMs are managed by KVM and Linux with asymmetric CPU time allocation.
- Four instances of the benchmark are run in the four asymmetric VMs, one in each VM. The VMs are managed by KVM and Linux with asymmetric scheduling latency adjustment.

- Four instances of the benchmark are run in the four asymmetric VMs, one in each VM. The VMs are managed by KVM and Linux with both asymmetric CPU time allocation and asymmetric scheduling latency adjustment.
- Four instances of the benchmark are run in the four asymmetric VMs, one in each VM. The VMs are managed by KVM and Linux with asymmetric CPU time allocation, asymmetric scheduling latency adjustment and resource conflict resolver.

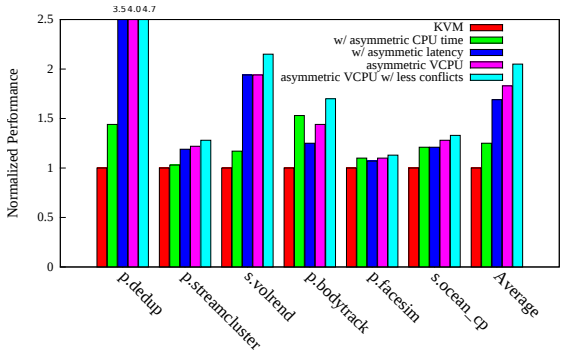

Figure 5.10 Performance Breakdown for the six benchmarks under the five scenarios

We breakdown the performance advantage of each component in the dynamic asymmetric VCPUs solution by comparing the benchmark performance under the above five scenarios. The reason to choose co-running identical workloads is that they have similar resource demands and would cause more resource conflicts, which can help us understand the advantage of resource conflict resolver in terms of reducing conflicts. The performance is normalized using the performance in the first scenario and presented in Figure 5.10.

As shown in the figure, the average performances with the five scenarios are 1, 1.25, 1.69 and 1.83, respectively. This indicates that all components contribute to the performance improvement. For some benchmarks like *p.dedup*, *p.streamcluster* and *volrend*, the asymmetric latency adjustment dominates the performance improvement. For some benchmarks like *bodytrack* and *facesim*, the asymmetric CPU time allocation dominates the performance improvement. For *ocean_cp*, both asymmetric CPU time allocation and asymmetric latency adjustment contribute to optimize the performance, and higher performance is achieved when both components are

employed. and can increase the performance up to 12.3x with *p.dedup* compared to that in the SMP VM. Moreover, after resource is distributed in an asymmetric way, the resource conflict resolver can further boost the performance by 12%, which indicates that there do exists the resource conflicts and this component can effectively reduce the conflicts to improve application performance on asymmetric VMs.

5.6 Conclusion and Future Work

This chapter addresses the performance degradation problem and performance variation problem of multicore programs running on VMs with multiple VCPUs. The approach is to dynamically adjust the performance features of VCPUs (and thus their CPU time resource and their computing capacities) based on the features of the computations on the VCPUs. Experiments show its effectiveness in improving performance. An alternative approach would be moving computation tasks across VCPUs inside each VM based on the computing capabilities of VCPUs. Though this requires the modification of programs or guest OSs and addressing a few other challenges, e.g., detecting VCPU computing capability and controlling the overhead of migrating tasks, the approach has a few advantages in understanding the resource demand of workload and avoiding the resource conflicts between VCPUs. This approach can be complementary to the dynamic asymmetric VCPU solution. Exploring this approach and integrating it with the dynamic asymmetric VCPU design is our future work.

CHAPTER 6

CONCLUSION

With virtualization technology, delivering computing resource as a utility through the cloud has gone mainstream. In accordance to the growth of core counts in physical machines, the number of virtual CPUs in a virtual machine also increases steadily. In modern virtual machines with multiple virtual CPUs, multi-threaded applications are run to achieve high performance, leveraging the aggregated computing power of these virtual CPUs. However, virtualization technology has not evolved enough to effectively support multi-threaded applications to achieve high performance. The executions of multi-threaded applications on virtual machines suffer serious performance issues. This causes the waste of resources and leads to frustrating user experience.

To achieve high performance on modern virtual machines, the dissertation identifies key issues and factors that have impact on the performance of multi-threaded application, and optimizes virtualization at different layers. The dissertation mainly targets CPU resource, which is the most important computing resource determining application performance.

To identify and diagnose performance issues, we have conducted extensive experiments, focusing on the CPU virtualization overhead incurred by synchronization and communication. Our results show that, when the physical system is not over-subscribed, the executions of multi-threaded applications can be slowed down by over 150%, and, when the system is over-subscribed, the slow down can be as much as 6x. We reveal that the main causes for these problems include the overhead incurred by handling synchronization and communication and mis-handled CPU resource sharing upon synchronization and communication.

Targeting the performance issues caused by spin-based synchronization, we designed and implemented the APPLES framework, which makes effective utilization of hardware support in processors to effectively reduce excessive spinning. The design of APPLES addresses two challenges. First, in order to promptly detect and preempt VCPUs when they spin excessively, APPLES adjusts dynamically the VCPU spinning threshold, based on the measurement of execution efficiency. Second, APPLES carefully selects and schedules VCPUs based on a few heuristics. Our experiments show that APPLES can improve system throughput by as much as 49%.

Targeting the most important performance factor for multi-threaded applications – scalability, we have analyzed key system factors and program features that may impact application scalability on virtual machines. The analysis is based on redefining scalability from the perspective of the CPU resource utilization efficiency. We show that application scalability on virtual machines is largely determined by how CPU time allocated to the virtual machine can be efficiently utilized (high utilization leads to high scalability). Since applications may utilize CPU resource in a different way on virtual machines than they do on physical machines, applications show different scalability on virtual machines than on physical machines. In some cases, applications may achieve better scalability on virtual machines. We have investigated the potential to improve application scalability on virtual machines by improving CPU resource utilization.

Motivated by the analysis, we have developed an effective solution to improve the utilization of CPU resource and to eventually improve application scalability on virtual machines. The solution creates dynamic asymmetric VCPUs, which have computing capability matching the workload distributed to the VCPUs. We show that the solution can effectively improve both the performance of individual VMs and the overall system throughput.

BIBLIOGRAPHY

[1] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, San Jose, California, USA, 2006.

[2] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, Seattle, WA, USA, 2008.

[3] J. Fisher-Ogden, "Hardware support for efficient virtualization," *University of California, San Diego, Tech. Rep*, 2006.

[4] T. Friebel and S. Biemueller, "How to deal with lock holder preemption," *Xen Summit North America*, Boston, MA, USA, 2008.

[5] A. Dorofeev, "Co-scheduling SMP VMs in VMware ESX Server." [Online]. Available: communities.vmware.com/docs/DOC-4960

[6] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, Houston, Texas, USA, 2013, pp. 369–380.

[7] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[8] K. Nguyen, "APIC virtualization performance testing and iozone." [Online]. Available: https://software.intel.com/en-us/blogs/2013/12/17/apic-virtualization-performance-testing-and-iozone

[9] Intel, "Intel 64 and IA-32 architectures software developer's manual." [Online]. Available: ftp://download.intel.com/design/processor/manuals/253668.pdf

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

[11] VMware, 2013. [Online]. Available: http://www.vmware.com/resources/techresources/10205

[12] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2009)*, vol. 2011, Austin, Texas, USA, 2009.

[13] "Perf wiki." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[14] M. Tosatti, "A walkthrough on some recent KVM performance improvements." [Online]. Available: http://www.linux-kvm.org/images/e/ea/2010-forum-mtosatti_walkthrough_entry_exit.pdf

[15] X. Ding, P. Gibbons, and M. Kozuch, "A hidden cost of virtualization when scaling multicore applications," in *USENIX HotCloud 2013*, San Jose, CA, USA, 2013.

[16] M. Righini, "Enabling Intel® virtualization technology features and benefits," Intel, Tech. Rep., 2010.

[17] A. Theurer, "KVM and big VMs," 2012. [Online]. Available: http://www.linux-kvm.org/images/5/55/2012-forum-Andrew-Theurer-Big-SMP-VMs.pdf

[18] S. Siddha, "Multi-core and linux* kernel." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.5973

[19] X. Xiang, B. Bao, C. Ding, and K. Shen, "Cache conscious task regrouping on multicore processors," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, Ottawa, Canada, 2012, pp. 603–611.

[20] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: bare-metal performance for i/o virtualization," in *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, London, United Kingdom, 2012.

[21] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines." in *Virtual Machine Research and Technology Symposium*, San Jose, California, USA, 2004, pp. 43–56.

[22] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for smp vms?" in *Proceedings of the Sixth European Conference on Computer Systems (EuroSys 2011)*, Salzburg, Austria, 2011, pp. 257–272.

[23] 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 2006)*, Minneapolis, MN, USA, 2006, pp. 124–133.

[24] 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 Environments (VEE 2013)*, vol. 48, no. 7, Houston, TX, USA, 2013, pp. 191–200.

[25] AMD, "AMD64 architecture programmer's manual volume 2: System programming." [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf

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

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

[28] M. Grund, J. Schaffner, J. Krueger, J. Brunnert, and A. Zeier, "The effects of virtualization on main memory systems," in *Proceedings of the Sixth International Workshop on Data Management on New Hardware (SIGMOD 2010)*, Indianapolis, IN, USA, 2010, pp. 41–46.

[29] J. Han, J. Ahn, C. Kim, Y. Kwon, Y.-r. Choi, and J. Huh, "The effect of multi-core on hpc applications in virtualized systems," in *European Conference on Parallel Processing (Euro-Par 2010)*. Springer, Santiago de Compostela, Spain, 2010, pp. 615–623.

[30] C. A. Waldspurger, "Memory resource management in vmware esx server," in *Proceedings of the 5th ACM Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA, 2002.

[31] S. K. Barker, T. Wood, P. J. Shenoy, and R. K. Sitaraman, "An empirical study of memory sharing in virtual machines." in *USENIX Annual Technical Conference (USENIX ATC 2012)*, Boston, MA, USA, 2012, pp. 273–284.

[32] P. Luszczek, E. Meek, S. Moore, D. Terpstra, V. M. Weaver, and J. Dongarra, "Evaluation of the HPC challenge benchmarks in virtualized environments," in *European Conference on Parallel Processing (Euro-Par 2011)*, Santiago de Compostela, Spain, 2011, pp. 436–445.

[33] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt, "Minimal-overhead virtualization of a large scale supercomputer," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011)*, Newport Beach, California, USA, 2011, pp. 169–180.

[34] N. Chakthranont, P. Khunphet, R. Takano, and T. Ikegami, "Exploring the performance impact of virtualization on an hpc cloud," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom 2014)*, Singapore, 2014, pp. 426–432.

[35] D. Cerotti, M. Gribaudo, P. Piazzolla, and G. Serazzi, "End-to-end performance of multi-core systems in cloud environments," in *Computer Performance Engineering*. Springer, 2013, pp. 221–235.

[36] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu, "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks," in *2013 IEEE International Congress on Big Data (BigData Congress)*, Santa Clara, CA, USA, 2013, pp. 9–16.

[37] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS 2015)*, Philadelphia, PA, USA, 2015.

[38] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: split guest/hypervisor execution on multi-core," in *USENIX 3rd Workshop on I/O Virtualization (WIOV 2011)*, Portland, OR, USA, 2011, pp. 1–7.

[39] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE 2005)*, Chicago, IL, USA, 2005, pp. 13–23.

[40] R. K. T, "[patch rfc 1/1] kvm: Add dynamic ple window feature." [Online]. Available: https://lkml.org/lkml/2012/11/11/14

[41] R. Krčmář, "[patch v3 7/7] KVM: VMX: optimize ple_window updates to VMCS." [Online]. Available: https://lkml.org/lkml/2014/8/21/456

[42] S. D. Lowe, "Best practices for oversubscription of cpu, memory and storage in vSphere virtual environments." [Online]. Available: https://software.dell.com/whitepaper/best-practices-for-oversubscription-of-cpu-memory-and-storage-in-vsphe823172/

[43] K. Raghavendra, "Virtual CPU scheduling techniques for kernel based virtual machine (KVM)," in *2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM 2013)*, Bangalore, India, 2013, pp. 1–6.

[44] J. Zhang, Y. Dong, and J. Duan, "ANOLE: A profiling-driven adaptive lock waiter detection scheme for efficient mp-guest scheduling," in *2012 IEEE International Conference on Cluster Computing (CLUSTER 2012)*, Beijing, China, 2012, pp. 504–513.

[45] R. R. Branco and V. Henson, "ebizzy-0.3," 2013. [Online]. Available: http://sourceforge.net/projects/ebizzy/

[46] P. Russell and M. Nordstrom, "dbench - measure disk throughput for simulated netbench run," 2005. [Online]. Available: http://manpages.ubuntu.com/manpages/raring/man1/dbench.1.html

[47] I. Molnar and Z. Yanmin, "Latest version of hackbench," 2008. [Online]. Available: http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c

[48] C. Kolivas, "Kernbench v0.50," 2009. [Online]. Available: http://ck.kolivas.org/apps/kernbench/kernbench-0.50/

[49] C. Yu, L. Qin, and J. Zhou, "A lock-aware virtual machine scheduling scheme for synchronization performance," *The Journal of Supercomputing*, pp. 1–13, 2015.

[50] X. Song, J. Shi, H. Chen, and B. Zang, "Schedule processes, not VCPUs," in *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys 2013)*, Singapore, 2013, pp. 1:1–1:7.

[51] L. Zhang, Y. Chen, Y. Dong, and C. Liu, "Lock-Visor: An efficient transitory co-scheduling for MP guest," in *2012 IEEE 41st International Conference on Parallel Processing (ICPP 2012)*, Pittsburgh, PA, USA, 2012.

[52] B. Wang, Y. Cheng, W. Chen, Q. He, Y. Xiang, M. M. Hassan, and A. Alelaiwi, "Efficient consolidation-aware vcpu scheduling on multicore virtualization platform," *Future Generation Computer Systems*, vol. 56, pp. 229–237, 2016.

[53] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2014)*, Cambridge, UK, 2014, pp. 394–405.

[54] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin, "Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016)*, Chicago, Illinois, USA, 2016.

[55] H. Mitake, T.-H. Lin, Y. Kinebuchi, H. Shimada, and T. Nakajima, "Using virtual CPU migration to solve the lock holder preemption problem in a multicore processor-based virtualization layer for embedded systems," in *IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2012)*, Seoul, South Korea, 2012, pp. 270–279.

[56] K. Raghavendra, S. Vaddagiri, N. Dadhania, and J. Fitzhardinge, "Paravirtualization for scalable kernel-based virtual machine (KVM)," in *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM 2012)*, Dubai, United Arab Emirates, 2012.

[57] S. Kashyap, C. Min, and T. Kim, "Opportunistic spinlocks: Achieving virtual machine scalability in the clouds," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 1, pp. 9–16, 2016.

[58] B. Huang and M. Zhu, "Research on necessity of adjusting PLE configuration," in *2014 International Conference on Cloud Computing and Internet of Things (CCIOT 2014)*, Changchun, China, 2014.

[59] J. Shan, X. Ding, and N. Gehani, "APLE: Addressing lock holder preemption problem with high efficiency," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom 2015)*, Vancouver, Canada, 2015, pp. 242–249.

[60] L. Bin, "Enhancement for PLE handler in kvm." [Online]. Available: https://lkml.org/lkml/2014/3/3/356

[61] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry, "Decoupling contention management from scheduling," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*, Pittsburgh, PA, USA, 2010, pp. 117–128.

[62] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein, "Optimal strategies for spinning and blocking," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 21, no. 2, pp. 246–254, 1994.

[63] B. He, W. N. Scherer III, and M. L. Scott, "Preemption adaptivity in time-published queue-based spin locks," in *International Conference on High-Performance Computing (HiPC 2005)*, Goa, India, 2005, pp. 7–18.

[64] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki, "A new look at the roles of spinning and blocking," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN 2009)*, Providence, RI, USA, 2009, pp. 21–26.

[65] "Amazon EC2 instance types." [Online]. Available: https://aws.amazon.com/ec2/instance-types

[66] L. Cheng, J. Rao, and F. Lau, "vScale: automatic and efficient processor scaling for smp virtual machines," in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys 2016)*, London, UK, 2016, p. 2.

[67] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin, "Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016)*, Chicago, Illinois, USA, 2016, pp. 343–352.

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

[69] L. Yavits, A. Morad, and R. Ginosar, "The effect of communication and synchronization on amdahl's law in multicore systems," *Parallel Computing*, vol. 40, no. 1, pp. 1–16, 2014.

[70] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA 2010)*, Saint-Malo, France, 2010, pp. 362–370.

[71] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010)*, Vancouver, British Columbia, Canada, 2010, pp. 1–12.

[72] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[73] N. J. Gunther, S. Subramanyam, and S. Parvu, "A methodology for optimizing multithreaded system scalability on multi-cores," *arXiv:1105.4301*, 2011.

[74] T. Oh, H. Lee, K. Lee, and S. Cho, "An analytical model to study optimal area breakdown between cores and caches in a chip multiprocessor," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2009)*, Tampa, Florida, USA, 2009, pp. 181–186.

[75] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2010)*, Atlanta, Georgia, USA, 2010, pp. 225–236.

[76] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa, "Characterizing multi-threaded applications based on shared-resource contention," in *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2011)*, Austin, TX, USA, 2011, pp. 76–86.

[77] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2012)*, Salt Lake City, UT, USA, 2012, p. 51.

[78] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013)*, Prague, Czech Republic, 2013, pp. 379–391.

[79] D. M. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments." in *2013 USENIX Annual Technical Conference (USENIX ATC 2013)*, San Jose, CA, USA, 2013, pp. 219–230.

[80] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2007)*, San Jose, California, USA, 2007, pp. 200–209.

[81] 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 (PACT 2016)*, Haifa, Israel, 2016, pp. 287–297.

[82] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2014)*, Cambridge, UK, 2014, pp. 394–405.

[83] Y. Peng, S. Wu, and H. Jin, "Towards efficient work-stealing in virtualized environments," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2015)*, Shenzhen, Guangdong, China, 2015, pp. 41–50.

[84] "Amazon EC2 X1 instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/x1/

[85] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013)*, Prague, Czech Republic, 2013, pp. 379–391.

[86] 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 (PACT 2016)*, Haifa, Israel, 2016, pp. 287–297.

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

[88] D. Kim, H. Kim, N. S. Kim, and J. Huh, "vCache: Architectural support for transparent and isolated virtual llcs in virtualized environments," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO 2015)*, Waikiki, Hawaii, 2015, pp. 623–634.

[89] A. Gundu, G. Sreekumar, A. Shafiee, S. Pugsley, H. Jain, R. Balasubramonian, and M. Tiwari, "Memory bandwidth reservation in the cloud to avoid information leakage in the memory controller," in *Proceedings of the Third ACM Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2014)*, Minneapolis, MN, USA, 2014, p. 11.

[90] C. Bae, L. Xia, P. A. Dinda, and J. R. Lange, "Dynamic adaptive virtual core mapping to improve power, energy, and performance in multi-socket multicores," in *The 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012)*, Delft, the Netherlands, 2012, pp. 247–258.

[91] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell, "VM 3: Measuring, modeling and managing VM shared resources," *Computer Networks*, vol. 53, no. 17, pp. 2873–2887, 2009.

[92] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, "Cachescouts: Fine-grain monitoring of shared caches in cmp platforms," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, Brasov, Romania , 2007, pp. 339–352.

[93] O. A.-Y. Assaf Schuster, Liran Funaro, "Ginseng: Market-driven llc allocation," in *2016 USENIX Annual Technical Conference (USENIX ATC 2016)*, Denver, CO, USA, 2016, p. 295.

[94] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *Real-Time and Embedded Technology and Applications Symposium (RTAS 2017)*, Pittsburgh, PA, USA, 2017, pp. 211–222.

[95] V. Selfa, J. Sahuquillo, S. Petit, and M. E. Gomez, "A hardware approach to fairly balance the inter-thread interference in shared caches," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

[96] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*, Atlanta, Georgia, USA, 2016, pp. 33–47.

[97] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys 2015)*, Bordeaux, France, 2015, pp. 18:1–18:17.

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

[99] R. C. Chiang and H. H. Huang, "TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011)*, Seattle, WA, USA, 2011, p. 47.

[100] T. Harris, M. Maas, and V. J. Marathe, "Callisto: co-scheduling parallel runtime systems," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys 2014)*, Amsterdam, Netherlands, 2014, p. 24.

[101] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA 2007)*, San Diego, California, USA, 2007, pp. 186–197.

[102] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA 2008)*, Salt Lake City, UT, USA, 2008, pp. 123–134.

[103] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the intel® core™ i7 turbo boost feature," in *IEEE International Symposium on Workload Characterization (IISWC 2009)*, Austin, TX, USA, 2009, pp. 188–197.

[104] H. Kim, S. Kim, J. Jeong, and J. Lee, "Virtual asymmetric multiprocessor for interactive performance of consolidated desktops," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2014)*, Salt Lake City, Utah, USA, 2014, pp. 29–40.

[105] J. Shan, X. Ding, and N. Gehani, "Apples: Efficiently handling spin-lock synchronization on virtualized platforms," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 7, pp. 1811–1824, 2017.

[106] "The open source development laboratory, osdl - database test suite." [Online]. Available: http://old.linux-foundation.org/labactivities/kerneltesting/osdldatabasetestsuite/

[107] Linux, "cgroups." [Online]. Available: https://en.wikipedia.org/wiki/Cgroups

[108] "Transaction processing performance council, 'tpc-w'." [Online]. Available: http://www.tpc.org/tpcw