CHARACTERIZING AND OPTIMIZING THE PERFORMANCE OF VIRTUALIZED NETWORK SYSTEMS IN THE CLOUD

by

KUN SUO

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

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2019
ACKNOWLEDGEMENTS

When I wrote down these words, I still clearly remembered the time when I got my bachelor degree. I was wondering whether I should spend another five or even more years to get a Ph.D. degree. Time flies and many years have slipped by unnoticed. Now the destination of my Ph.D. study is close and it’s time for me to thank all the people who have companied, helped and encouraged me through this journey.

First of all, I would like to thank my advisor, Dr. Jia Rao. It was my great honor to be the first Ph.D. student of Dr. Rao. From my personal perspective, Dr. Rao is the best advisor and mentor I ever met: He is young and full of energy; cultivated and prestigious. He guided the students from the big research ideas to the small design details; helped the students from career development to normal daily life. He leads our lab to the forefront of the computer system academics. Without the guide and help from Dr. Rao, it is impossible for me to arrive here and have today’s achievements.

I would also like to thank the dissertation committee members, Dr. Hong Jiang, Dr. Song Jiang, and Dr. Jiang Ming, for their help and encouragement throughout my Ph.D. study at UT Arlington. I am filled with gratitude to them.

I am also grateful to every labmate I ever met and worked with. Not matter in the snowy Colorado or in the warm Texas, I am very lucky to have them around discussing the difficulties we faced and sharing happiness with each other. Our lab is like another big family for each of us coming from different corners of world. I would like to thank Yanfei Guo, Dazhao Cheng, Rui Zhao, Yong Zhao, Wei Chen, Shaoqi Wang, Xiaofeng Wu, Fan Ni, Xingbo Wu, Xingsheng Zhao, Haitao Wang, Mengxiao Wang, Zhichao Yan. All of my labmates made my Ph.D. journey so colorful and full of fun and happiness. I am honored to grow up with them together and also hope everyone having a successful career and a happy life in the future.

I would like to thank the best friends in my life, my parents. My parents made me who I am and they tried everything they could, day by day, year over year, to provide the unyielding support and help me achieve my success. This thesis is dedicated to them for their constant support, encouragement and belief in my dreams.

While they are no longer with me, I know my paternal and maternal grandmother are watching. My debt to them is beyond measure and I miss them this moment. This thesis is also dedicated to them. I would also like to thank my other relatives, uncles, aunts, brothers and sisters. Their care and love accompany me through my growth. I am so grateful to have each family member in such a beautiful and kind family.
During my Ph.D. journey, there are also so many people who have provided me with help and support but I do not even know their names. I cannot write them here but I express my gratitude to them. Thank you.

June 10, 2019
ABSTRACT

CHARACTERIZING AND OPTIMIZING THE PERFORMANCE OF VIRTUALIZED NETWORK SYSTEMS IN THE CLOUD

KUN SUO, Ph.D.
The University of Texas at Arlington, 2019

Supervising Professor: Jia Rao

To leverage the elastic resource allocation of cloud computing and enhance the service availability and productivity, numerous applications and businesses have been moved from the traditional data centers into the cloud during the past decade. Despite the benefits introduced by virtualization, such as high resource utilization, flexible resource management and operation cost reduction, it also incurs additional overhead, scheduling delays as well as semantic gaps among hardware, operating system and applications. These issues can cause non-negligible impact to the performance and quality-of-service (QoS) of the cloud applications, especially for I/O-intensive services. Meanwhile, the increasing scale and complexity of the cloud infrastructure aggravate the above problems, making both characterization and optimization of the virtualized network performance much more difficult.

In this dissertation, we explore the potential opportunities in various cloud infrastructures including the traditional virtual machines (VM), emerging containers as well as the application runtime, and present lightweight and efficient approaches to characterize and optimize the network performance in virtualized environments.

From the perspective of characterization in virtualized networks, we designed and developed an in-band packet profiler, namely *Time Capsule*, which traced packet level granularity latency across different boundaries in virtualized systems with negligible overhead. By leveraging the superpower of extended Berkeley Packet Filter (eBPF), we proposed *vNetTracer*, a highly efficient and programmable profiler to trace application network performance in virtualized networks. Both *Time Capsule* and *vNetTracer* shed light on the virtualized network monitoring and can help users analyze, identify and localize potential issues inside virtualized networks. Besides the above tools, we also investigated the existing virtualized networks inside the containers and performed a comprehensive study of representative container networks. Our study illustrated many important findings that could help users select the appropriate network for their workloads and guide the optimization of the existing container networks.
From the perspective of optimization in virtualized networks, we proposed and developed several approaches for improving the performance of I/O-intensive applications in virtualized environments. We investigated the suboptimal I/O performance of applications in the VMs or containers, and discovered the CPU discontinuity caused priority inversions and rendered existing I/O prioritization in the guest OS ineffective. Thus, we proposed xBalloon, a lightweight approach to preserving static and dynamic priorities between I/O-bound and compute-bound tasks and boosting the I/O performance under discontinuous time. We also presented an in-depth performance analysis of the Java virtual machine (JVM) and found the existing design of JVM incurred inefficient garbage collection, which introduced the low throughput and long tail latency of the applications. To mitigate the overhead, we propose a number of solutions to these issues, including enforcing GC thread affinity to aid multicore load balancing and designing a more efficient work stealing algorithm. We implemented and evaluated the above techniques in both local lab clusters and public cloud with representative benchmark suites. The experiment results and case studies demonstrated the effectiveness of our proposed techniques. The main contribution of this dissertation lied in simple yet effective solutions that characterize and improve the network performance in various virtualized environments of the cloud.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ................................................................. iii
ABSTRACT .................................................................................. v

Chapter | Page
--- | ---
1. INTRODUCTION ................................................................. 1
  1.1 Characterizing Network Packet Latency with Lightweight and Cross Boundary Tracing .......................................................... 2
  1.2 Understanding Virtualized Network Activities through Efficient and Programmable Monitoring .................................................. 3
  1.3 Bridging the Gap between Physical and Virtual Environments in I/O Prioritization ................................................................. 3
  1.4 Characterizing and Optimizing Parallel Garbage Collection on Multicore Systems ................................................................. 4
  1.5 Dissertation Organization ...................................................... 5
2. RELATED WORK ................................................................. 6
  2.1 Characterizing the Networking Activities and System Performance ................................................................. 6
    2.1.1 Latency Measurement and Analysis ......................................... 6
    2.1.2 Monitoring Schemes and Tracing Across Boundaries .................. 6
    2.1.3 Logs Based Monitoring and Dynamic Instrumentation ................. 7
  2.2 Improving Virtualized Infrastructure Efficiency .................................. 8
    2.2.1 Bridging the Semantic Gaps .................................................. 8
    2.2.2 Optimizing the Critical I/O Path .............................................. 8
    2.2.3 Reducing VM Scheduling Delays .............................................. 8
  2.3 Application Runtime Optimization ............................................. 9
    2.3.1 Optimizing Garbage Collection .............................................. 9
    2.3.2 Accelerating Java Applications ............................................. 10
3. TIME CAPSULE: TRACING PACKET LATENCY ACROSS DIFFERENT LAYERS IN VIRTUALIZED SYSTEMS ......................... 11
  3.1 Introduction ................................................................. 11
  3.2 Design and Implementation .................................................. 12
  3.3 Overhead and Optimizations ................................................ 16
  3.4 Evaluation ................................................................. 18
    3.4.1 Overhead Analysis .......................................................... 18
    3.4.2 Per Packet Latency .......................................................... 19
    3.4.3 Latency Decomposition .................................................. 19
3.4.4 Case Studies .................................................. 21
3.5 Discussions ..................................................... 23
3.6 Conclusion ....................................................... 23

4. VNETTRACER: EFFICIENT AND PROGRAMMABLE PACKET TRACING
IN VIRTUALIZED NETWORKS ........................................... 25
4.1 Introduction ..................................................... 25
4.2 Background ...................................................... 27
4.3 vNetTracer Design .............................................. 29
   4.3.1 Overview ................................................... 29
   4.3.2 Tracing Across Boundaries ................................ 30
   4.3.3 Efficiency ................................................... 32
   4.3.4 Programmability ............................................ 34
   4.3.5 Implementation ............................................ 35
4.4 Evaluation ...................................................... 35
   4.4.1 Evaluation Settings ........................................ 36
   4.4.2 Overhead Analysis .......................................... 36
   4.4.3 Case Study I: Network Delay in the Open vSwitch ....... 37
   4.4.4 Case Study II: Tuning the Scheduler in Hypervisors ....... 40
   4.4.5 Case Study III: Bottlenecks of the Container Architecture 42
4.5 Conclusion ...................................................... 45

5. XBALLOON: PRESERVING I/O PRIORITIZATION IN VIRTUALIZED OSES 46
5.1 Introduction ..................................................... 46
5.2 Background and Motivation .................................... 49
   5.2.1 Assumptions ................................................ 49
   5.2.2 I/O Prioritization in Linux ................................ 49
   5.2.3 Time Discontinuity ....................................... 49
   5.2.4 Degraded I/O Performance due to CPU Discontinuity .... 50
5.3 Analyzing Priority Inversions .................................. 51
   5.3.1 Short-term Priority Inversion ............................. 51
   5.3.2 Long-term Priority Inversion .............................. 52
5.4 xBALLOON Design .............................................. 53
   5.4.1 Differential Clocks ........................................ 54
   5.4.2 CPU Balloon ............................................... 54
   5.4.3 Semi-Work-Conserving Scheduling ....................... 54
   5.4.4 Enforcing Static Priority ................................ 57
   5.4.5 Preserving Dynamic Priority .............................. 57
5.5 Implementation .................................................. 60
5.6 Evaluation ...................................................... 60
CHAPTER 1
INTRODUCTION

During the past decade, the digital world is experiencing the greatest transition in history from traditional data centers to the cloud. This transition happens not only for hundreds of thousands of companies who migrate numerous applications and businesses into the cloud to leverage the elastic cloud resource and enhance their service availability and productivity, but also in everybody’s personal devices which fundamentally change their ways of working, living, entertainment, etc. Even though the cloud infrastructure brings massive benefits including high resource utilization, flexible resource management as well as operation cost reduction, it also introduces additional issues such as virtualization overhead, scheduling delays and the semantic gaps among hardware, operating system and applications at the same time. Unfortunately, these issues cannot be ignored as they perform huge impact on the performance and quality-of-service (QoS) of the cloud applications, especially for those I/O-intensive services. Meanwhile, the increasing cloud scale and its complexity also aggravate the above problems, making both characterization and optimization of the virtualized network performance much more difficult.

In this dissertation, we propose inventing advanced tools to efficiently characterize and understand the fundamental causes of the issues in the cloud virtualized networks. Based on the tracing data and experiment analysis, we further explore more efficient and smarter solutions which integrate the hypervisor, operating systems and applications to deliver better virtualized network performance and cloud system utilization. From the perspective of characterization in virtualized networks, we designed and developed several tools to trace the network activities in virtualized systems with negligible overhead. These tools shed light on the virtualized network monitoring and can help users analyze, identify and localize potential issues inside virtualized networks. From the perspective of optimization, we proposed and developed approaches for improving the performance of I/O-intensive applications in various virtualized environments, such as the virtual machines in Xen or KVM, the Docker containers and the Java virtual machines (JVM). All of these attempts not only provide better QoS guarantees for the end users, but also improve the resource utilization as well as the system efficiency for the cloud providers. In summary, our efforts include:

• Time Capsule—an in-band profiler to provide fine-grained and across boundaries latency tracing in virtualized networks with acceptable overhead [141],...
• vNetTracer—a highly efficient and programmable profiler that traces network activities in the virtualized systems [145],
• xBalloon—a lightweight approach to preserving I/O prioritization between I/O- and compute-bound applications during discontinuous time [146],
• An in-depth performance analysis of garbage collection (GC) in HotSpot JVM and corresponding optimizations which enforces GC thread affinity to aid multicore load balancing and accelerates work stealing [142].

In the following of this chapter, I will introduce the motivation and the contributions of each work, then outline the rest of this dissertation.

1.1 Characterizing Network Packet Latency with Lightweight and Cross Boundary Tracing

As increasing number of applications are running or built on top of virtualized systems, which have complex I/O stacks across different layers, it makes difficult to monitor per-packet latency in network applications, let alone the identification of the root causes of long latencies. Monitoring the virtualized network activities has some unique challenges compared to the traditional physical systems. Virtualization not only introduces additional hardware and software boundaries, such as software stacks and protection domains, but also increases workload consolidation and system utilization. All the above makes lightweight and efficient packet latency tracing much more difficult. Existing solutions and tools either identify performance anomalies by analyzing distributed logs and stitching them based on unique IDs, which introduce non-negligible runtime overhead and cannot guarantee provide useful information, or only provide limited tracing within certain scopes.

To address the above challenges, we designed Time Capsule [141], an in-band packet profiler that traces per-packet latency across different layers or protection domains in virtualized systems. By leveraging the packet itself, we timestamps packets at predefined tracepoints and embeds the timing information into the payload of the packet to achieve cross-boundary latency tracing. We also devise a set of optimizations, such as reducing the instrumentation cost and collecting tracing data in shared memory, to reduce the runtime overhead of Time Capsule. With such the design, Time Capsule is able to measure fine-grained latency, e.g., packet level, across protection domains in virtualized systems with negligible overhead. Besides, Time Capsule is transparent to cloud users and does not require any changes to traced applications. Experiments have demonstrated that Time Capsule decomposes and attributes network latency to various layers in virtualized I/O stack, which can help identify latency bottlenecks, localize potential bugs, and motivate optimizations.
1.2 Understanding Virtualized Network Activities through Efficient and Programmable Monitoring

As the scale of cloud computing increases and enormous kinds of applications and services emerge on the cloud, virtualized network is becoming increasingly vital to support them for high and predictable performance. Compared to traditional network infrastructure, virtualized network provides many advantages including dynamic deployability, high management flexibility and scalability. However, virtualized network also involves additional complex infrastructure and changes dynamically, making application network monitoring, performance abnormalities diagnosing as well as root causes identifying, much more difficult. In order to trace network activities, many works have been proposed based on the system logs. These tools either adopt static log analysis or use machine learning on logs to troubleshoot the system issues. However, analyzing massive logs not only introduces non-negligible runtime overhead, but also cannot accommodate to the dynamic virtualized network and monitoring requirements. Many other state-of-the-art works either introduce additional annotations and middleware layer in virtualized systems or need to involve significant modification to the network stack. All these attempts change the existing systems which not only incur additional overhead but also cannot be flexibly programmed for various demands.

To address the above problems, we design vNetTracer [145] to enable flexible and programmable network performance monitoring for virtualized networks in a highly efficient manner. To enable cross-boundary tracing, vNetTracer adds a unique ID in individual network packet and the packet ID is embedded into the header of the packet such that it is carried over the boundaries of domains. By leveraging the extended Berkeley Packet Filter (eBPF), vNetTracer provides efficient performance monitoring and rich metrics which support customized network tracing for different requirements at runtime. We used vNetTracer to trace different networking activities in the virtualized systems and the case studies showed that vNetTracer can effectively measure the virtualized network and satisfy different scenarios. Take network latency as an example, with the help of vNetTracer, we can trace the packet latency of target application, attribute network latency to various components of the virtualized network stack and further locate the potential bottleneck inside the systems.

1.3 Bridging the Gap between Physical and Virtual Environments in I/O Prioritization

Poor I/O performance has been a longstanding issue in virtualized systems, especially for workloads that have overlapping I/O and computation. Literature has believed that the suboptimal I/O performance is due to virtualization overhead and inefficient virtual
machine scheduling and thus proposed approaches bypassing the virtualization layer and prioritizing I/O processing at the hypervisor. However, we find another fundamental cause of the performance suboptimality. In virtualized environments, CPU sharing and capping are widely adopted to improve resource utilization and performance isolation. Such the CPU multiplexing breaks the continuous CPU time into discrete intervals, in which guest operating systems are scheduled to run. The discontinuity renders existing I/O prioritization in the guest OS ineffective, leaving a large room for performance improvement. Due to such the time discontinuity, we identify two priority inversion issues in guest OS CPU scheduling. First, time discontinuity can render in-guest CPU accounting inaccurate, leading to short-term priority inversions. Second, the work-conserving scheduling in guest OS, which is designed for continuous time, allows low priority tasks to consume resources that would otherwise be utilized by high priority tasks in the future, causing long-term priority inversions. The above priority inversions will significantly degrade the I/O-intensive application performance and increase the system unpredictability.

We argue that the resource management policies in the guest OS should be adapted to virtualized and multi-tenant environments to efficiently utilize virtual resources. To achieve that, we propose xBalloon [146], a lightweight approach to preserving static and dynamic priorities between I/O-bound and compute-bound tasks. xBalloon addresses the priority inversion issues by making the guest OS aware of the discontinuous time and adapting the guest CPU scheduling to truly differentiate processes even in the presence of resource sharing at the hypervisor. It centers on two designs: i) a CPU balloon process in the guest OS to represent the time gaps that are not allocated to the VM; ii) an integration of the balloon process into CFS scheduling to preserve static and dynamic priorities. Since I/O requests arrive individually at discrete intervals, the guest OS decides if low priority compute tasks should run during the idle period based on the CPU availability in the future. If low priority tasks should not execute, the balloon process is scheduled to run, which pauses the VM. As such, the VM executes under non-work-conserving mode to save CPU time for future high priority I/O tasks. If the VM has sufficient resources, i.e., the CPU allocation is higher than the I/O demand, the balloon process is suspended, switching back to work-conserving mode. Our results have demonstrated that xBalloon is effective in boosting I/O performance and preserving task priorities both in a local Xen environment and on Amazon EC2.

1.4 Characterizing and Optimizing Parallel Garbage Collection on Multicore Systems

Java is becoming increasingly popular and a growing number of modern computing systems, such as Hadoop, Spark, Kafka, Cassandra, Android, etc., and widely used services,
such as Google Web Search, Gmail, Google Docs, etc., are built on Java virtual machine (JVM). One major component of JVM is the Garbage Collection (GC), which performs the memory management automatically and developers can simply reply on it and do not need to care the allocation and deallocation of memory. GC is usually prohibitively expensive as many studies have shown that GC could take a non-trivial portion of application execution time. Many factors lead to the low efficiency of JVM GC including the unnecessary memory accesses, GC lock contentions, etc. We characterize the performance of the OpenJDK 1.8 and present an in-depth analysis on the behavior of Parallel Scavenge (PS), a state-of-the-art and the default garbage collector in the HotSpot JVM. We found that the GC thread and task load imbalance as well as the inefficient task stealing might degrade the performance of JVM Garbage Collection, especially on the multicore system.

To address the above inefficiency, we propose techniques and best practices on the OpenJDK [142]. Specifically, in order to reduce the long GC time caused by the load imbalance, we find that the thread-pinning and task-affinity resolve much of such vulnerability and implement our own load balance functions and interfaces. To manage the delay due to the inefficient GC stealing algorithms, we analyze the existing stealing mechanism in GC and propose an adaptive stealing policy that dynamically adjusts the number of steal attempts based on the number of active GC threads. We put the aforementioned observations and optimizations together to evaluate the potential savings from the vanilla OpenJDK JVM. Our experiment results show that each of the optimizations contributes to the GC improvement as well as the overall performance improvement of many micro-benchmarks and real-world big data workloads.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2 we discuss the related work of characterizing or optimizing the network performance in the virtualized systems. In Chapter 3 we introduce the challenges to trace tail latency in virtualized systems and present an in-band packet profiler, Time Capsule, that traces packet level latency across different boundaries in virtualized systems with low overhead. In Chapter 4 we present vNetTracer, an efficient and programmable packet profiler to monitor the network activities in virtualized systems. In Chapter 5 we present how xBalloon addresses static and dynamic priorities inversion between I/O- and compute-bound tasks under discontinuous time. In Chapter 6 we present that the unfair lock acquisitions in the JVM and interactivity between the Linux CFS scheduler and JVM will hurt the GC efficiency as well as scalability, and propose optimizations such as annotating GC tasks with scheduling hints and a more efficient work stealing algorithm. In Chapter 7 we summarize our contributions and discuss the future work.
CHAPTER 2
RELATED WORK

2.1 Characterizing the Networking Activities and System Performance

2.1.1 Latency Measurement and Analysis

Much effort has dedicated to analyzing factors that affect network latency. DARC [150] proposes runtime latency analysis to find main latency contributors. Li et al. [97] explore the potential causes for tail latency on multi-core machines. Similarly in virtualized environment, Soroban [138] studies the latency in VMs or containers by using machine learning and Xu et al. [164] introduce a host-centric solution for improving latency in the cloud. As Software Defined Network (SDN) and Network Function Virtualization (NFV) become popular in recent years, latency measurement and analysis play more important roles in modern networks. For example, relying on accurate timestamps provided by SoftNIC [76], DX [94] proposes a new congestion control scheme to reduce queuing delay in datacenters. However, no of the above studies focus on tracing packet-level granularity latency across boundaries in the cloud systems.

2.1.2 Monitoring Schemes and Tracing Across Boundaries

There are in general two ways to monitor performance and trace program execution in complex systems. Non-intrusive tracing systems [40, 45, 55, 115, 129, 133, 163] leverage existing application logs and performance counters to diagnose performance issues and detect bugs. Annotation-based monitoring [53, 46, 63, 137] offers users more flexibility to selectively trace certain components in the monitored systems. However, both approaches still face the challenges of balancing the tradeoff between the effectiveness of tracing and its overhead. In comparison, dynamic instrumentation allows flexible logging and tracing to be installed dynamically. For instance, Pivot Tracing [106] leverages aspect-oriented programming to export variables for dynamic tracing and designs a query languages to selectively invoke user-defined tracepoints. Similarly, our work Time Capsule [141] also requires tracepoints to be manually inserted by administrator but supports a flexible selection of tracepoints and varying sampling rates.

Tracing tools are widely used in complex systems to diagnose performance problems. However, many tools are limited within certain boundaries. For example, gperf [12] is an application-level tool to analyze UNIX program performance while Systemtap [31],
DTrace [49], and Perf [19] are used to trace source code or collect performance data inside Linux kernel. Similarly, Xentrace and Xenalyze [33] can only be used to trace event in the Xen hypervisor. To address the limitations, many efforts have been proposed to trace beyond boundaries. Stardust [149] uses breadcrumb record associated with application requests to correlate events in separate components or computers. Whodunit [51] adopts synopsis, a compact representation of a transaction context, to profile transactions across distributed machines. Pivot tracing [106] uses a per-request container, baggage, to correlate logging information with a particular request context. There are also other works that infer the relationship between events in distributed environments [46, 63, 53, 149, 51, 137, 126, 127]. For instance, Appinsight [126] and Timecard [127] try to locate the latency bottleneck for mobile networks and their idea is to add time information into the packet during the transmission.

2.1.3 Logs Based Monitoring and Dynamic Instrumentation

Many tools or monitoring systems [11, 19, 40, 45, 129, 133] provide non-intrusive tracing, which leverages existing application logs and performance counters, to diagnose performance issues and detect bugs in complex systems. However, tracing based on existing system logs or static tracepoints cannot guarantee providing what the users are interested in. In addition, too many logs might also introduce lots of overhead and overwhelm the valuable information. Statistical approaches, e.g., data mining, machine learning [55, 115, 138, 163, 151], are promising directions towards automated identification of the issues inside the systems based on the massive tracing logs, the execution environment as well as the application performance. For example, Chow et al. proposed the Mystery Machine [55], which analyzed traces of over 1.3 million requests collected over 30 days in Facebook and generated the system dependency model and calculated the critical path. However, such solutions are still based on existing system logs or tracing information, which lack of the flexibility for satisfying the dynamic user requirements and service changes, especially in virtualized networks.

Annotation based monitoring [46, 53, 63, 137] allows users to selectively trace the systems based on their purposes. For example, Pip [128] provided application level annotation monitoring while Pinpoint [53] and X-Trace [63] added annotations in the libraries and middleware software. However, such approaches not only are still lack of the flexibility in the dynamic monitoring, but also face the challenges of balancing the tradeoff between the tracing efficiency and its overhead. Dynamic instrumentation allows logging and tracing to be installed dynamically and flexibly. Pivot Tracing [106] leveraged aspect-oriented programming to export variables for dynamic tracing and designed a query languages to selectively invoke user-defined tracepoints. However, Pivot Tracing is designed specifically for Hadoop and is used to trace distributed applications only in user space. Instead, our
work vNetTracer [144, 145] focuses on dynamic tracing through the entire virtualized network stack. Many other system tools, such as SystemTap [31], DTrace [49], also support dynamic instrumentation. However, SystemTap introduces much overhead for high frequency tracing, which is not suitable for virtualized network instrumentation. DTrace is a troubleshooting tool in the OSes like Solaris, FreeBSD, etc., and it does not support system tracing in the Linux.

2.2 Improving Virtualized Infrastructure Efficiency

2.2.1 Bridging the Semantic Gaps

The semantic gap in virtualized environments has been a well-recognized issue. There were studies on bridging the gaps for VM introspection [59, 64, 132], adapting TCP congestion control [54], guest and hypervisor coordination [170], optimizing virtualized I/O scheduling [92, 119], inferring information about process [86] and buffer management [85] inside VMs; however they focused on exposing VM information to the hypervisor to aid bare-metal resource management. Such designs not only make the hypervisor more complex but also make enforcing fairness between tenants difficult. In contrast, our work xBalloon focuses on exposing the information on resource allocation at the hypervisor to the guest OS to make wise scheduling decisions inside VMs. xBalloon improves I/O performance in VMs without violating the autonomy VM’s resource management or seeking any algorithmic changes at the hypervisor for I/O performance optimization.

2.2.2 Optimizing the Critical I/O Path

As virtualization introduces additional layers of abstraction, indirection, and data movement, the critical I/O path in virtualized systems contains excessive asynchrony and latency [141]. To reduce virtualization overhead, existing work optimizes the critical I/O path by offloading performance critical operations from VMs to the hypervisor, such as TCP ACK generation and congestion control [65, 89], TCP segmentation/checksum calculation [110], and device driver functionalities [111], or by packet coalescing [112]. Other work proposed more aggressive changes to I/O virtualization via exit-less interrupt delivery [71] and I/O path reconstruction [66, 84, 107]. xBalloon is complementary to these approaches and guarantees critical I/O operations to be scheduled timely in the guest OS.

2.2.3 Reducing VM Scheduling Delays

The VM scheduling delay prevents I/O operations from VMs being timely processed. To address this issue, many efforts, like shortening time slices [43, 162], preemptive
scheduling [171], partial boosting [92, 136] or dedicating CPUs [161] are proposed. There are also other work focusing on optimizing scheduling for MapReduce clusters [88], data center workloads [165] and I/O interrupt handling on SMP VMs [54]. VM scheduling delay can also be alleviated by using smaller time slices in the hypervisor. There are two drawbacks to use small time slices. First, while I/O workloads benefit from short time slices due to more responsive scheduling, CPU-bound workloads could suffer performance degradation because frequent context switching causes loss of data locality. Second, the change of VM time slice at the hypervisor will affect all users. It may be undesirable for some users. In contrast, xBalloon proposed in this dissertation allows the guest OS to autonomously decide how to use its CPU. Our work is most related to task-aware VM scheduling [92], which prioritizes VMs doing I/O and de-schedules them once the hypervisor detects the VMs are performing computation. xBalloon offers several advantages over this approach. First, it precisely preserves the static and dynamic priorities. Second, xBalloon does not make algorithmic changes to a hypervisor’s core scheduling algorithm.

2.3 Application Runtime Optimization

2.3.1 Optimizing Garbage Collection

Many studies [52, 62, 78, 103, 125, 148, 156, 157, 167] have demonstrated that it is effective to improve Java application performance by reducing the GC overhead. For instance, Yu et al. [167] discovered unnecessary memory accesses and calculations during the compaction phase of a full GC and proposed an incremental query model for reference calculation to accelerate the GC. To address deficiencies in the existing JVMs, many researchers [70, 98, 103, 135, 148] proposed specialized garbage collectors or re-designed the JVM runtime. Some other works [56, 118] proposed new interfaces to facilitate the interaction between applications and garbage collectors. Such designs require changes to the application source code. In this dissertation, we identified a previous unknown performance issue due to the lack of coordination between parallel GC and the underlying OS. We proposed two optimizations in HotSpot, which are transparent to user programs.

Memory locality and access balance on NUMA multicore machines attracted much attention in recent years. Gidra et al. [68] studied the scalability of throughput-oriented GCs and proposed to map pages and balance GC threads across NUMA nodes. NumaGiC [69] focused on improving memory access locality without degrading the GC parallelism. It avoided costly remote memory accesses and restricts GC threads to collect its own node’s memory. While these works employed node affinity to aid NUMA-aware memory management, our work uses thread affinity to avoid harmful interactions between the JVM and the OS scheduler. Sartor et al. [134] found that most benchmarks did not benefit from thread
pinning in the Jikes Research Virtual Machine (RVM) [131]. This dissertation discovered a performance issue due to unfair locking in the HotSpot JVM and Linux CFS.

Work stealing dynamically balances tasks among threads to improve GC efficiency [68, 69, 78, 124, 156]. Gidra et al. [68, 69] proposed to restrict work-stealing to GC threads that run on the same node. Wessam [78] introduced an optimized work-stealing algorithm that uses a single thread to accelerate the termination phase. Wu et al. [156] proposed task-pushing instead of work-stealing to improve stealing efficiency. Qian et al. [124] replaced the steal_best_of_2 design with a heuristic-based stealing algorithm. A GC thread immediate aborts stealing if failed and only steals from the same victim if stealing was successful. Although this algorithm reduced the number of failed steal attempts, it undermined concurrency during work stealing. In contrast, we proposed to optimize the existing best_of_2 random stealing algorithm and accelerate the termination phase by adaptively altering the termination criteria based on the number of active GC threads.

2.3.2 Accelerating Java Applications

Existing studies [125, 158, 160] improved Java performance by adopting an efficient scheduler. For instance, Qian et al. [125] found that the FIFO scheduler in Linux achieved higher GC efficiency and mutator performance due to less heap competition and fewer context switches. A recent work [101] found several bugs in CFS that cores are left idle when there are runnable threads, and proposed fixes. The GC inefficiency identified in this dissertation is caused by the lack of coordination between the JVM and Linux CFS, and thus the fixes had no effect on the thread stacking issue.

As big data applications are becoming increasingly popular, many researchers start to study optimizations of data-intensive workloads on distributed systems through improved memory management [70, 102] or JVM runtime [98, 105, 103, 116, 117]. Another trend in optimizing Java performance is to coordinate with hardware architectures. Studies [52, 68, 69, 134] explored NUMA-aware designs to improve GC locality and application performance. Maas et al. [104] proposed a hardware-assisted GC which had high throughput and good memory utilization to overcome the STW pause. Hussein et al. [83] proposed a GC-aware governor balance on-chip energy consumption and the performance. These studies are orthogonal to our dissertation.
CHAPTER 3
TIME CAPSULE: TRACING PACKET LATENCY ACROSS DIFFERENT LAYERS IN VIRTUALIZED SYSTEMS

Latency monitoring is important for improving user experience and guaranteeing quality-of-service (QoS). Virtualized systems, which have complex I/O stacks spanning multiple layers and often with unpredictable performance, present more challenges in monitoring packet latency and diagnosing performance abnormalities compared to traditional systems. Existing tools either trace network latency at a coarse granularity, or incur considerable overhead, or lack the ability to trace across different boundaries in virtualized environments. To address this issue, we propose Time Capsule (TC), an in-band profiler to trace packet level latency in virtualized systems with acceptable overhead. TC timestamps packets at predefined tracepoints and embeds the timing information into the payload of these packets. TC decomposes and attributes network latency to various layers in the virtualized network stack, which can help monitor network latency, identify bottlenecks, and localize potential issues.

3.1 Introduction
As virtualization has become mainstream in data centers, a growing number of enterprises and organizations are moving applications into the cloud, such as Amazon EC2 [2]. It is well believed that virtualization introduces significant and often unpredictable overhead to I/O-intensive workloads, especially latency-sensitive network applications. To improve user experience and guarantee QoS in the cloud, it is necessary to efficiently monitor and diagnose the latency of these applications in virtualized environments.

However, compared to traditional systems, it is more challenging to trace I/O workloads and troubleshoot latency problems in virtualized systems. First, virtualization introduces additional software stacks and protection domains while traditional tracing tools designed for physical machines are unable to trace across the boundaries (e.g., between the hypervisor and the guest OS). Second, workload consolidation inevitably incurs interference between applications, which often leads to unpredictable latencies. Thus, fine-grained tracing of network latency, instead of coarse-grained monitoring, is particularly important to guaranteeing QoS. Third, many applications in the cloud are highly optimized and the tracing tool should incur negligible performance impact [137]. Otherwise, tracing overhead
may hide seemingly slight but relatively significant latency changes of the cloud services. Last, as clouds host diverse workloads, it is prohibitively expensive to devise application specific tracing mechanisms and this calls for a system level and application transparent tracing tool.

There exist many studies focusing on system monitoring and diagnosing. Traditional tools, such as SystemTap [31], DTrace [49], Xentrace [39], or existing instrumentation systems, such as DARC [150] and Fay [61], are limited to use within a certain boundary, e.g., only in the hypervisor or in a virtual machine (VM). None of them can trace activities, e.g. network processing, throughout the entire virtualized I/O stack. Many state-of-the-art works, like Mystery Machine [55], Draco [90], LogEnhancer [168], lprof [169], identify performance anomalies among machines based on distributed logs. However, analyzing massive logs, not only introduces non-negligible runtime overhead, but also cannot guarantee providing the information users need, e.g., statistics on tail latency. In addition, such out-of-band profiling needs additional effort to stitch the distributed logs [149, 51], and time drift among logs on different machines may also affect the accuracy.

In this work, we propose Time Capsule (TC), a profiler to trace network latency at packet level in virtualized environments. TC timestamps packets at predefined tracepoints and embeds the timing information into the payloads of packets. Due to in-band tracing, TC is able to measure packet latency across different layers and protection boundaries. In addition, based on the position of tracepoints, TC can decompose and attribute network latency to various components of the virtualized network stack to locate the potential bottleneck. Further, TC incurs negligible overhead and requires no changes to traced applications. We demonstrate that fine-grained tracing and latency decomposition enabled by TC shed light on the root causes of long tail network latency and help identify real performance bugs in Xen.

3.2 Design and Implementation

The challenges outlined in Section 3.1 motivate the following design goals of Time Capsule: (1) cross-boundary tracing; (2) fine-grained tracing; (3) low overhead; (4) application transparency. Figure 4.2 presents a high-level overview of how TC enables tracing for network send (Tx) and receive (Rx). TC places tracepoints throughout the virtualized network stack and timestamps packets at enabled tracepoints. The timing information is appended to the payload of the packet. For network receive, before the traced packet is copied to user space, TC restores the packet payload to its original size and dumps the tracing data to a kernel buffer, from where the tracing data can be copied to user space for offline analysis. For network send, trace dump happens before the packet is transmitted by the physical NIC. Compared to packet receive, we preserve the timestamp of the last
tracepoint in the payload of a network send packet to support tracing across physical machines. Since tracepoints are placed in either the hypervisor, the guest kernel or the host kernel, TC is transparent to user applications. Next, we elaborate on the design and implementation of TC in a Xen environment.

**Clocksource** To accurately attribute network latency to various processing stages across different protection domains, e.g., Dom0, DomU, and Xen, a reliable and cross-domain clocksource with high resolution is needed. The para-virtualized clocksource xen meets the requirements. In a Xen environment, the hypervisor, Dom0, and the guest OS all use the same clocksource xen for time measurement. Therefore, packet timestamping using the xen clocksource avoids time drift across different domains. Next, the clocksource xen is based on the Time Stamp Counter (TSC) on the processor and has nanosecond resolution. It is adequate for latency measurement at the microsecond granularity. A similar clocksource kvm-clock is also available in KVM. Furthermore, we enabled constant_tsc and nonstop_tsc of Intel processors, which can guarantee that TSC rate is not only synchronized across all sockets and cores, but also is not affected by power management on individual processors. As such, TSC ticks at the maximum CPU clock rate regardless of the actual CPU running speed. For cross-machine tracing, the clocks on physical nodes may inevitably tick at different rates due to different CPU speeds. Therefore, the relative difference between timestamps recorded on separate machines does not reflect the actual time passage. Figure 3.2 shows the relationship between the TSC readings on two machines with different CPU speeds. The slopes of the two lines represent the maximum CPU frequency on the respective machines. There exist two challenges in correlating the timestamps on separate machines. First, TSC readings are incremented at different rates (i.e., different slopes). Second, TSC registers are reset at boot time or when resuming from hibernation. The relative difference between TSC readings on two machines includes the absolute distance of these machines since last TSC reset. For example, as shown in Figure 3.2, the distance
between TSC reset on two machines is denoted by $\alpha = |t_{\text{reset}}^a - t_{\text{reset}}^b|$, where $t_{\text{reset}}^a$ and $t_{\text{reset}}^b$ are the last TSC reset time of machine $a$ and $b$, respectively.

**Tracepoints** are placed on the critical path of packet processing in the virtualized network stack. When a target packet passes through a predefined tracepoint, a timestamp based on local clocksource is appended to the packet payload. The time difference between two tracepoints measures how much time it spent in a particular processing stage. For example, two tracepoints can be placed at the backend in Dom0 and frontend in DomU to measure packet processing time in the hypervisor. As timestamps are taken sequentially at various tracepoints throughout the virtualized network stack, TC does not need to infer the causal relationship of the tracepoints (as Pivot Tracing does in [106]) and the timestamps in the packet payload have strict happened-before relation.

**Cross-machine tracing** requires that the varying TSC rates and reset times on different machines be taken into account for accurate latency attribution. Specifically, timestamps recorded on separate machines should be calibrated to determine the latency due to network transmission between machines. We illustrate the TSC calibration process in Figure 3.2.

Assume that a packet is sent from machine $a$ (denoted by the dotted blue line) at time $t_1$, which has a faster CPU and its TSC starts to tick earlier, and received at time $t_2$ on machine $b$ (denoted by the solid red line) with a slower CPU. Without TSC calibration, the difference $t_{\text{sc}}^b - t_{\text{sc}}^a$ can show negative transmission time. There are two ways to measure packet transmission time in the network based on the two timestamps $t_{\text{sc}}^a$ and $t_{\text{sc}}^b$ taken at the sender and receiver machines. First, the difference of TSC reset time $\alpha$ can be estimated as $|\frac{t_{\text{sc}}^a}{\text{cpu freq}_a} - \frac{t_{\text{sc}}^b}{\text{cpu freq}_b}|$, where $t_{\text{sc}}^a_{\text{sync}}$ and $t_{\text{sc}}^b_{\text{sync}}$ are the instantaneous TSC readings on the two machines at exactly the same time. This can be achieved through distributed clock...
synchronization algorithms which estimate the packet propagation time in a congestion-free network and adjust the two TSC readings. Once $\alpha$ is obtained, the absolute TSC difference $\beta$ is calculated as $\beta = \alpha \times \text{cpu freq}_a$. Then, the first calibration step is to derive $tsc'_a = tsc_a - \beta$ to remove the absolute TSC difference. As shown in Figure 3.2, $tsc'_a$ is the TSC reading of packet transmission at the sender if machine $a$ resets TSC at the same time as machine $b$. Further, the equivalent TSC reading at the receiver machine $b$ when the packet starts transmission is $tsc'_b = tsc'_a \times \frac{\text{cpu freq}_b}{\text{cpu freq}_a}$. Finally, the packet transmission time is the difference between the timestamps of packet send and receive on the receiver machine $b$: $t_2 - t_1 = \frac{tsc_b - tsc'_b}{\text{cpu freq}_b}$.

The first calibration method only requires the examination of one packet to measure packet transmission time but relies on an accurate estimation of $\alpha$. Since $\alpha$ is constant for all packet transmissions between two particular machines, an alternative is to estimate network condition based on the comparisons of multiple packet transmissions. Similar to [94], which compares packet transmission time with a reference value in a congestion-free environment to estimate network congestions, we can roughly measure packet transmission time as $\frac{tsc_a}{\text{cpu freq}_a} - \frac{tsc_b}{\text{cpu freq}_b}$ and use cross-packet comparison to identify abnormally long transmission time. However, this method only identifies relative transmission delays with respect to a reference transmission time, which is difficult to obtain in production datacenter network and may be variable due to packet being transmitted through different routes.

**Tracing payload** To enable tracing latency across physical or virtual boundaries, TC adds extra payload to a packet to store the timestamps of tracepoints. Upon receiving a packet at the physical NIC or copying a packet from user space to kernel space for sending, TC uses `__skb_put(skb, SIZE)` to allocate additional space in the original packet. The tracing information is removed from packet payload and dumped to a kernel buffer before a packet is copied to the application buffer in user space or sent out by the physical NIC.
Figure 3.3 shows the structure of a TC-enabled packet. The tracing payload contains two types of data: the tracing raw data and the tracing metadata. The tracing raw data consists of 8-byte entries, each of which stores the timestamp of a tracepoint. Users can place plenty of tracepoints in the virtualized network stack based on their needs and choose which tracepoints to enable for a particular workload. The tracing metadata uses the annotation bits to indicate if the corresponding tracepoint is enabled or not (1 as enabled). Users define an event mask to specify the enabled tracepoints and initialize the tracing metadata. The size of the tracing payload depends on the number of enabled tracepoints. For latency tracing across VMs on the same physical machine, packet is transferred between the hypervisor and domains through shared memory. The packet size is not limited by the maximum transmission units (MTUs). Thus, TC is able to allocate sufficient spaces in packet payload for tracing without affecting the number of packets communicated by the application workloads. For tracing across different physical machines, we dump all the timestamps before packets are sent out by the NIC but preserve the last timestamp recorded at the sender side (the sender side raw data in Figure 3.3) in the tracing payload. When the packet arrives at the receiver machine, new tracing data will be added after the sender side raw data. As such, the tracing data pertaining to the same packet stored on multiple machines can be stitched together by the shared timestamp.

### 3.3 Overhead and Optimizations

Despite the benefits, tracing at packet level can introduce considerable overhead to network applications. For highly optimized services in the cloud, such tracing overhead can significantly hurt performance. In this section, we discuss the sources of tracing overhead and corresponding optimizations in Time Capsule.

**Time measurement** It usually incurs various levels of cost to obtain timestamps in the kernel space. If it is not properly designed, fine-grained tracing can significantly degrade network performance, especially increasing packet tail latency.

**Optimization** We compare the cost of different clocksource read functions available in various domains using a simple clock test tool [6] and adopt `native_read_tscp` to read from the xen clocksource at each tracepoint. Compared to other time functions, which incur an overhead ranging from several microseconds to a few milliseconds, the function `native_read_tscp` adds only about 20 nanoseconds latency at a tracepoint. The overhead is negligible compared to tens and hundreds of microseconds latency in a typical network request.

**Trace collection** The dump of traces to storage is the most expensive operation in TC. If the completion of each packet triggers a disk write, the overhead will be prohibitively high. Further, the overhead grows as the intensity of network traffic increases.
Figure 3.4: Time Capsule incurs negligible overhead to packet latency with various (a) number of tracepoints and (b) sampling rates. The sampling rate is set to 1/4 in (a) and the number of tracepoints is 10 in (b).

**Optimization** We adopt a ring buffer in the guest network stack (receiver side) and the NIC driver in the driver domain (sender side) to temporarily store the tracing data. Each time the tracing payload is removed from a packet, the tracing data is copied to the buffer. The latest trace sample over-writes the oldest data if the circular buffer is full. We use `mmap` to map the kernel buffer to user space `/proc` file system, from where the traces can be dumped to persistent storage. We design a user space trace collector that periodically dumps and clears collected traces. As trace collection happens infrequently and can be performed offline on another processor, it does not hurt the network latency.

**Instrumentation cost** Too much instrumentation is one of the common issues in system monitoring, especially for latency-sensitive applications. As Google’s tracing infrastructure Dapper [137] shows, over-sampling increases the average latency of web search by as much as 16.3%, while just inflicting a marginal 1.48% drop on throughput.

**Optimization** We devise two optimizations to reduce instrumentation overhead. First, TC only selectively traces packets from a targeted applications. We use a combination of IP address and port number to identify the packets that should be traced. Other network packets skip the tracepoints and are processed normally. Second, as Figure 3.3 shows, tracing metadata in each packet allows TC to configure a flexible tracing rate (i.e., via the sampling decision bit) and to select a subset of tracepoints (i.e., via the annotation bit). The sampling decision bit enables tracing for a portion of packets in a network traffic based on a user-defined sampling rate. The annotation bit determines which tracepoint(s) should be enabled for a particular packet.
3.4 Evaluation

Experimental Setup Our experiments were performed on two PowerEdge T420 servers, connected with Gigabit Ethernet. Each server was equipped with two 6-core 1.90GHz Intel Xeon E5-2420 CPUs and 32GB memory. The host ran Xen 4.5 as the hypervisor and Linux 3.18.21 in the driver domain. The VMs in the host had a single virtual CPU (vCPU) and 4GB memory, and ran Linux 3.18.21 as the guest OS.

3.4.1 Overhead Analysis

First, we analyze the overhead of Time Capsule on network latency. Figure 3.4 plots the average, 99th, and 99.9th percentile latency of UDP packets using Sockperf [28]. We varied the number of tracepoints and the sampling rate. As shown in Figure 3.4, the number of tracepoints does not have much impact on packet latency, with no more than 1.5% latency increase (ten tracepoints) compared to that without TC (zero tracepoint). Similarly, the performance impact due to different sampling rates is insignificant. A sampling rate of 1 increased packet latency by 2% compared to that with a sampling rate of 1/16. As discussed in Section 3.3, TC’s overhead mainly comes from timestamping the packet at tracepoints and manipulating packet payload, which takes tens of nanoseconds at each tracepoint. Given that typical network applications have latency requirements in the range of hundreds of microseconds to a few milliseconds, TC’s overhead is negligible even with tens of tracepoints. In real systems, if enabling a large number of tracepoints for fine-grained tracing raises overhead concerns, lowering the packet sampling rate is likely to provide adequate trace data for network-intensive workloads. In summary, TC adds negligible overhead to network latency and overhead can be controlled by varying the number of tracepoints and sampling rate.

Figure 3.5: Compared to coarse-grained tracing, packet level tracing provides more information about user-perceived latency.
3.4.2 Per Packet Latency

User-perceived latency is an important QoS metric. However, it is difficult to track individual user experiences in virtualized environments. When application-level per-request logging is not available, system-wide tracing can be effective in identifying performance issues of individual users. Next, we demonstrate that packet level tracing reveals problems that could be hidden in coarse-grained tracing.

We created a scenario in which users-perceived latency suffered sudden hike due to a short network burst from another application. We chose Sockperf as the application under test and used Netperf [22] to generate the interfering traffic. Figure 3.5 plots the per-second average latency and packet level latency of Sockperf. The burst arrived at the 1.5th second and left at the 6.5th second. We have three observations from Figure 3.5 (b): i) packet level latency accurately captured latency fluctuations during the burst; ii) packet level latency measurement timely reflected user-perceived latency immediately after the network spike left (at 6.5s); iii) most importantly, packet level tracing successfully captured a few spikes in latency around the 11th second. The latency spike may be due to the backlogged packets during the traffic burst.

In contrast, per-second average latency (as shown in Figure 3.5 (a)) hid performance fluctuations, was not responsive to load changes, and missed the important latency issue after the burst traffic left. Microbursts, the sudden and rapid traffic bursts in the network, can cause long tail latency or packet loss and may not be captured by coarse-grained monitoring. TC enables packet level latency tracing and can effectively identify slight performance changes. Next, we show that TC further decomposes the packet latency into time spent in different stages to help locate the root causes of long latency.

3.4.3 Latency Decomposition

A detailed breakdown of packet latency sheds light on which processing stage in the virtualized network stack contributes most to the overall latency and help identify abnormalities
Figure 3.7: Latency decomposition when Sockperf is co-located with HPCbench in the same VM.

at certain stages. Figure 3.6 and Figure 3.7 show the latency and its breakdown of Sockperf under two scenarios: i) the VM hosting Sockperf alone; ii) the VM hosting Sockperf and HPCbench [14] simultaneously. In scenario ii, two separate clients sent Sockperf UDP requests and HPCbench UDP stream flows, respectively. While it is expected that the co-location of Sockperf and HPCbench in the same VM causes interference to Sockperf, we show that latency breakdown provides insight on how to mitigate the interference.

Figure 3.6 (a) and (b) show the receiver side latency for 500 packets and the latency breakdown when Sockperf ran alone in the VM. Without interference, the latency stabilized at about 60 $\mu$s and the processing at the driver domain, the hypervisor, and the guest kernel contributed equally to the overall latency. In contrast, Sockperf latency degraded by up to 35x and became wildly unpredictable when co-running with throughput-intensive HPCbench workload, as shown in Figure 3.7 (a). Note that although interference also exists on physical machines when co-locating latency-sensitive workloads with throughput-intensive workloads, the performance degradation is not as drastic as that in virtualized environments. Latency breakdown suggests that most degradation was from prolonged processing in Dom0.

To pinpoint the root cause, we added two additional tracepoints to further break down packet processing in Dom0: i) packet processing in Dom0’s network stack (denoted as Dom0 (a)) and ii) processing in the VM’s backend NIC driver at Dom0 (denoted as Dom0 (b)). As shown in Figure 3.7 (b), most time in Dom0 was spent in the backend NIC driver. After an analysis of Dom0’s backend driver code, we found that the excessively long latency was due to batching the memory copy between the backend and frontend drivers. The backend driver does not copy packets to a VM until the receive queue of the physical NIC is depleted. All received packets will be copied to a VM in a batch to amortize the cost of memory copy. This explains why workloads with bulk transfer degrade the performance of latency-sensitive applications. A large number of packets from throughput-intensive workloads fill up the receive queue in the backend driver, preventing the packets
of latency-sensitive application from being transferred to the VM. The analysis based on latency decomposition suggests that limiting the batching size in the backend driver would alleviate the interference.

### 3.4.4 Case Studies

With packet level tracing and latency decomposition, TC helps associate the excessively long latency to certain processing stages in the virtualized network stack. In this section, we describe our discovery of multiple bugs in Xen’s credit scheduler with the help of TC. These scheduler bugs cause long tail latency of I/O workloads. We reported the bugs to the Xen community and they were confirmed by engineers from Citrix [35, 36, 4]. Next, we explain how TC helped locate these bugs.

Xen’s credit scheduler is designed to prioritize I/O-bound VMs while not compromising the overall system utilization and fairness. Thus, if an I/O-bound VM consumes less than its fair CPU share, it always has a higher priority than a CPU-bound VM and I/O performance should not be affected by co-located CPU-bound workloads. Figure 3.8 shows the performance of Sockperf when its host VM shared the same physical CPU with another CPU-bound VM. We observed that Sockperf latency degraded significantly due to the interference from the CPU-bound VM. For approximately every 250 packets, latency grew to as high as 30 ms and started to descend until the next spike. Latency decomposition in Figure 3.8 (b) shows that latency spikes always started with long delays in Xen, which dominated the overall latency. This indicates that the latency spike started with packets being blocked in Xen and then the delay propagated to Dom0. The delay in Xen was close to 30 ms, which matched the length of default time slice in the credit scheduler. These observations gave a hint that the long latency is correlated with the VM scheduler.

The analysis led to the discovery of the first bug in the credit scheduler: Xen mistakenly boosts the priority of a CPU-bound VM, thereby preventing an I/O-bound VM
Figure 3.9: Bug-2 [36]: Xen does not timely activate I/O-bound VMs that are deactivated due to long idling. Bug-3 [4]: I/O-bound VMs’ BOOST priority can be prematurely demoted.

from being prioritized. If this happens, the I/O-bound VM needs to wait for a complete time slice (i.e., 30 ms) before the CPU-bound VM is descheduled by Xen due to the expiration of its time slice. The latency breakdown in Figure 3.8 (b) further shows that the first wave of packets in the latency spike were copied to the grant table in Xen but cannot be processed by the I/O-bound VM because the VM was not scheduled to run. Thus, the delay is attributed to the wait time in Xen. After the grant table was full but the I/O VM was not yet scheduled, new coming packets stayed in the backend driver of Dom0, which explained the propagation of delay to Dom0. After fixing this bug, both the average and tail latency were greatly improved.

Using the same methodology, we discovered another two bugs in the credit scheduler that also contributed to the long tail latency issue. The reasons behind excessively long tail latency are usually complicated. As packets with long latency are ephemeral and difficult to be captured by coarse-grained monitoring, finding the causes is even harder. Figure 3.9 shows the occurrence of abnormal latency in twenty thousands packets. Only 5% of the packets suffered long latency and their occurrence had no clear patterns. As the magnitude and occurrence of the abnormal latency were unpredictable, using TC traces alone was not enough to identify the bugs. As Figure 3.9 (b) shows similar patterns of the long latency – long delays in Xen followed by delays in Dom0, we separately instrumented Xen to report its scheduling events and performed side-by-side comparisons of the Xen trace and TC packet trace. We used event and packet timestamps to correlate the Xen scheduling events with abnormal packets. Considerable manual effort was needed to identify two additional bugs in the credit scheduler. Through the above examples, we have demonstrated that TC is quite useful to identify, localize and analyze latency issues, especially for the long tail latency.
3.5 Discussions

Packet size TC relies on the additional space in the packet payload to store tracing information. For small packets and the packets transferred between VMs on the same host, TC’s tracing payload does not increase the number of packets needed by the original application. In the case of cross-machine tracing, an 8 byte space is needed for the last timestamp from the sender. It is possible that this will increase the number of MTUs transferred by the original application. When network is congested, the additional MTUs could be the source of overhead. For packets larger than the MTU, NIC that supports GSO/GRO features will split the large packets into separate MTUs at the sender side. TC only needs to append an 8-byte timestamp to the last MTU.

Dynamic instrumentation Currently, we manually add tracepoints in TC and Dom0 and DomU kernels need to be recompiled to use the new tracepoints. Dynamic instrumenting the Linux kernel is possible by using the extended Berkeley Packet Filter (eBPF). However, dynamic instrumentation cannot manipulate packet payload, thereby unable to cross the boundaries between Dom0, DomU and the hypervisor.

Automated analysis Although TC provides detailed information about packet processing in virtualized systems, considerable manual effort is needed to identify the causes of unsatisfactory performance. TC can only correlate the stages on the critical path of I/O processing with the overall network performance. For performance issues due to other resource management schemes, such as VM scheduling and memory allocation, a causal analysis of TC trace and other system traces is necessary. Statistical approaches that find the correlation between the traces are promising directions towards automated identification of the root causes.

Extending TC to disk I/Os TC leverages the commonly shared data structure skb to associate tracing information with individual packets. Virtualized disk I/O is as complex as virtualized network I/O and often suffers poor and unpredictable performance. However, it is more challenging to trace disk I/O across multiple block I/O stacks in virtualized systems. The challenges are the lack of a shared data structure between the protection domains to pass the tracing information and aggressive optimizations of disk I/O at different layers of the virtualized system.

3.6 Conclusion

Latency is a critical QoS factor for network applications. However, it is challenging to monitor latency in virtualized systems. This work presents Time Capsule (TC), an in-band packet profiler that traces packet level granularity latency across different boundaries in virtualized systems. TC incurs negligible overhead to network performance and does not
need any changes to applications. We demonstrated that the fine-grained packet tracing and cross boundary latency decomposition enabled by TC shed light on the latency monitoring and helped us identify three design bugs in a Xen environment. Guided by TC, we are able to trace, quantify, and analyze issues that cause long latency in virtualized systems.
CHAPTER 4
VNETTRACER: EFFICIENT AND PROGRAMMABLE PACKET TRACING IN VIRTUALIZED NETWORKS

As the scale of cloud systems continues to grow, virtualized networks that provide connectivity between services within and across data centers, are becoming increasingly important to the performance and reliability of the cloud. Despite many advantages, including fast deployment, ease of management, and programmability, virtualized networks require additional layers of abstraction and complicate monitoring and diagnosis of performance issues compared to traditional networks on physical hardware. Virtualized networks usually connect components in multiple protection domains, such as a guest OS, the hypervisor, network bridges, and separate virtualized network functions. There is no efficient means to trace packet transmission across the boundaries. Furthermore, it is challenging to reason about the performance of dynamic virtualized networks. Therefore, fine-grained, user customizable, and reconfigurable network tracing becomes a great need. To address these challenges, we proposed vNetTracer, an efficient and programmable packet profiler for virtualized networks. vNetTracer relies on the extended Berkeley Packet Filter (eBPF) to dynamically insert user-defined trace programs into a live virtualized network without any changes to the applications or restarts of the monitored network. Through three case studies, we demonstrate the effectiveness of vNetTracer in diagnosing various virtualized networking problems.

4.1 Introduction

The adoption of virtualization in enterprise systems and data centers has given rise to on-demand, elastic, and cost-effective cloud services. Virtualized networks, which provide connectivity to physically or virtually isolated virtual machines (VMs) or containers, are critical to horizontally scaling the cloud services. Studies have shown that network virtualization techniques, such as software defined network (SDN) and network function virtualization (NFV), can improve network utilization while offering better quality-of-service (QoS) guarantees [75, 77, 93]. However, the additional layers of abstraction, high resource consolidation and complexity in the virtualized networks make it difficult to understand, diagnose, and optimize networking performance in the cloud.
Unlike the conventional networks, virtualized networks present unique challenges to performance tracing. First, virtualized networks usually span multiple protected domains, such as the host OS or hypervisor, virtual devices and the virtual OS. Tracing end-to-end performance requires that events within each domain can be correlated. However, no tools can efficiently cross the boundaries of the protected domains to associate the tracing information. Second, performance issues of virtualized networks usually occur with high load, in which the networking performance is sensitive to the tracing overhead. This requires a lightweight tracing tool which can monitor virtualized networks with negligible expenditure. Third, the complexity and volatility of virtualized networks require that the tracing tool is reconfigurable in real time and provides a rich set of metrics for performance diagnosis.

There are a plethora of works focusing on tracing network and distributed systems. However, they fall short of addressing the challenges in tracing virtualized networks. To make sense the performance of virtualized networks, such as understanding the causes of long tail latency under high load, it is necessary to trace the network applications at the packet level. Existing studies have shown that recording the tracing data per packet, which often requires significant data copy and context switching between kernel space and user space, incurs prohibitive overhead during the system monitoring [41, 84]. Furthermore, virtualized networks often comprise multiple layers of abstraction to attain isolation and allow for reconfiguration. Thus, it is essential to instrument the virtualized system to provide the needed trace for performance diagnosis. Manual or static instrumentations [46, 53, 55, 61, 63, 137, 141, 169] often require intrusive changes to the system and cannot be generalized to tracing different applications. Machine learning-based log analysis [90, 115, 163] relies on comprehensive instrumentation of the system, from which meaningful information can be mined. Existing dynamic instrumentation tools, such as DTrace [9], SystemTap [31], and DARc [150], cannot trace across the boundary of protected domains in virtualized systems. Distributed tracing systems, e.g., Pivot tracing [106], are usually implemented at application or middleware level, thereby unable to trace packet transmission inside OS kernels.

In this work, we leverage the extended Berkeley Packet Filter (eBPF), a dynamic tracing mechanism in modern Linux kernels, to enable lightweight and programmable tracing for virtualized networks on multiple nodes. Although eBPF has been increasingly adopted for traffic control [25], network security [48] or accelerating network infrastructure [15, 34, 42], there are no prior explorations of system performance tracing based on eBPF. To this end, we present the design and implementation of vNetTracer, an eBPF-based tracing framework, which enables efficient, flexible and end-to-end network performance monitoring for applications in virtualized networks. Different from the traditional tracing tools, vNetTracer has the following features:
• **Tracing across boundaries**: vNetTracer enables end-to-end tracing across boundaries and can correlate distributed events in separate, protected domains.

• **Efficiency**: vNetTracer incurs marginal runtime overhead and is efficient for performance monitoring and troubleshooting in the highly consolidated and optimized virtualized networks.

• **Programmability**: vNetTracer provides rich performance monitoring metrics, supports customized network packet tracing, and can be configured based on different requirements. Users can modify tracepoints, tracing rules or actions in vNetTracer at runtime.

To achieve the above goals, we make three contributions in designing vNetTracer. First, to enable the end-to-end tracing across software or hardware boundaries, vNetTracer generates a unique trace ID for each packet and embeds the ID into the network packet header of the target application. The trace ID is used to differentiate individual network packet and construct the tracing log for further analysis. Second, we develop a set of performance metrics based on the tracing data, which characterize the performance of virtualized networks, including per-flow throughput, the decomposition of end-to-end latency, per-flow packet drop rate, per-device network processing time, etc. Last, we make several optimizations to minimize the runtime overhead during the network tracing.

Another contribution of this work is the use of vNetTracer to trace network performance in various virtualized systems and case studies show that vNetTracer can effectively monitor virtualized networks and satisfy different scenarios. Specially, vNetTracer helps us find that 1) the throughput-intensive flow in the ingress port of Open vSwitch (OVS) might cause the network congestion and delay latency-intensive flow through OVS; 2) the default configuration of the Xen’s credit2 scheduler incurs long tail network latency when executing CPU-intensive VMs and latency-intensive VMs on the same physical CPU; 3) the inefficient processing of a large number of softirqs in multicore systems imposes a significant performance bottleneck for container overlay networks.

### 4.2 Background

**eBPF based tracing.** The classic Berkeley Packet Filter (BPF) [109] is a kernel architecture for packet capture, which permits sending and receiving network packets at data link layers. However, due to its limited instruction set and difficulty in programming, the classic BPF is only used in few applications, e.g., tcpdump. Extended BPF (eBPF) is an extension of classic BPF, which introduces lots of new features and improves the performance. For instance, eBPF introduces new in-kernel Just-In-Time (JIT) machine, more register support and many new data structures for generating more complex and advanced eBPF programs. As shown in Figure 4.1, eBPF allows programmers to attach user-defined programs into
the kernel and the compiled eBPF bytecode can be executed on a live in-kernel VM, which performs insignificant negative impact to the kernel. Once the tracing events are triggered, the monitoring data can be either temporarily stored in the eBPF data structures inside kernel or collected asynchronously to the user space. Linux started to support eBPF since kernel 3.15 and introduced more BPF enhancements in the later versions.

Compared to the traditional monitoring, tracing based on the eBPF provides several advantages. First, it enables users to trace high frequency modules, such as context switches or packet processing, with little runtime overhead. Second, instead of adding inflexible and dull log inside the systems, the eBPF tracing is highly programmable and can be designed for different purposes. Last, as the tracing logics can be loaded or unloaded dynamically, it does not involve too many modifications to the existing systems.

**eBPF versus SystemTap.** SystemTap [31] is a tracing platform which is used for dynamically instrumenting processes and Linux kernel activities. Many previous efforts have analyzed the SystemTap runtime overhead [24, 32, 60, 72]. In general, the overhead of SystemTap comes from two aspects. First, the frequency of traces and the actions that SystemTap script performs have a significant impact on the instrumentation overhead [72]. For instance, tracing high performance network I/O, which processes tens of thousands interrupts each second, might have non-negligible overhead to the monitored systems. Second, the compilation of the SystemTap script during the start stage or tracing data collection between kernel and user space during the finish stage might also incur some overhead. In comparison, eBPF programs execute through an efficient virtual machine inside the kernel and the JIT compiling minimizes the execution overhead of the eBPF code. We will further compare and discuss the overhead of SystemTap with eBPF in Section 4.4.
Limitation. vNetTracer relies on recent eBPF features, and consequently requires a Linux kernel 4.9 and above. Also, the eBPF program is limited by its size, which allows at most 4k instructions. In addition, as the eBPF programs are only attached at tracepoints such as network sockets, kprobes, etc., vNetTracer is also limited by the tracepoint position. In comparison, many other techniques, such as Time Capsule [141], can add tracepoints arbitrarily inside the virtualized network. Furthermore, although vNetTracer is transparent to the network applications, we still need to modify the kernel in order to accurately trace each packet. Thus, vNetTracer is not completely transparent to the entire system. However, as the discussed in Section 4.3, vNetTracer only involves tens of lines of code modification inside the kernel.

4.3 vNetTracer Design
4.3.1 Overview

Figure 4.2 illustrates the architecture of vNetTracer. The key components of vNetTracer include a control data dispatcher, an agent on each monitoring machine and a raw data collector.

The control data dispatcher executes on the master node. It reads the user input and generates formatted configuration files in control packages and tracing scripts. Then the dispatcher sends the files to agents on remote monitoring machines. The agent receives the configured files from the dispatcher and executes eBPF programs at defined locations of the configuration files on the monitoring nodes. Agents collect the
tracing data based on the rules in the configuration files and then send the collected data to a centralized raw data collector.

The raw data collector also executes on the master node. It collects the raw tracing data from the agents and performs offline analysis based on the tracing data.

Next, we used a concrete example to describe how vNetTracer works. Suppose we need to measure the network latency between two Virtual eXtensible LAN (VXLAN) layers in the multiple host container network. We use flannel$_i$ to represent the VXLAN network device on the $i_{th}$ node. First, we input the following information into the control data dispatcher to generate formatted control package: (1) the filter rules, such as the containerized application source IP, destination IP, source port, destination port, etc; (2) the tracepoint information, including device name flannel$_i$, device ID, etc; (3) the action that records the current system time in nanosecond; (4) the global information like the database configuration, table names. Next the control data dispatcher sends the customized tracing scripts to remote tracing agents on the $i_{th}$ node. All eBPF scripts are attached to device flannel$_i$ and execute the time record action when the targeted network packets pass through. Network packets which do not match the tracing rules will not be traced. Once the raw tracing data is fetched, it is stored locally and then gathered to the database on the master node. After the data is collected, further operations such as data cleaning or calculation can be done for analysis. In this example, we calculate the time from flannel$_i$ to flannel$_j$ to get network latency between two VXLAN devices.

### 4.3.2 Tracing Across Boundaries

Tracepoints provide the system level entry point for vNetTracer to attach customized source code to instrument the system. In the current design, vNetTracer supports instrumenting kernel functions, return of kernel functions, kernel tracepoints and raw sockets through kprobe, kretprobe, tracepoints and network devices. Application monitoring could be traced through user level tracepoints such as uprobe and uretprobe. The location of a tracepoint is defined and enabled through user configuration files. Whenever execution of
Figure 4.4: Calculating the clock drift between two machines under the Cristian’s algorithm [57].

In order to trace across boundaries, vNetTracer distinguishes the network applications through their IP address and port number in the packet header, and identifies individual network packets by adding a unique ID. The packet ID is embedded into the header of the packet such that it is carried over the boundaries of domains. As shown in Figure 4.3, for the TCP packets, we use a 4-byte space in the options of the TCP header. For UDP packets, we use __skb_put() to allocate a 4-byte additional space to the original packet at the sender side. We generate a 32-bit random number as the packet ID and store it in the space when the packet is copied from user space to kernel space. The UDP packet ID is then removed from the packet payload through pskb_trim_rcsum() before it was copied to the application buffer in the receiver side to guarantee the application transparency. As the above additional operations only involve tens of nanoseconds overhead, they do not harm the microsecond level application latency. Besides the unique packet ID, vNetTracer also records the packet number, packet length and current system time for the detailed network measurement when a packet goes through the tracepoints, which is further discussed in Section 4.3.4.

In order to measure network performance metrics such as throughput, latency, etc., we need to get timestamps from a high resolution clock source. In each tracing script, we obtain the nanosecond-level granularity time record from the function bpf_ktime_get_ns(). This function reads the clock source CLOCK_MONOTONIC inside Linux kernel, which cannot be set by users and represents monotonic time since the kernel is booted. The nanosecond resolution of CLOCK_MONOTONIC is adequate for both network throughput measurement at second granularity and latency measurement at microsecond granularity. In addition, as the function bpf_ktime_get_ns() is executed inside the tracing script as backend and such a process always runs in the kernel space, there is almost no
overhead to read time from the clock source, and no kernel and user space context switches happen during the above process.

The clocks on different physical or virtual nodes may inevitably have time skew for cross-machine tracing in distributed systems. To mitigate this issue, we adopt Cristian’s algorithm [57] and measure the relative clock skew between the master node and the monitoring nodes. As depicted in Figure 4.4, we attach two tracing scripts at the NIC interfaces of master node and monitoring node. We record the timestamps once the packets were sent or received from the interfaces. On the client side, the round trip time $T_{RTT}$ is measured as $T_4 - T_1$. On the server side, the processing time $T_{pro}$ is measured as $T_3 - T_2$. Thus, the one way transmission time $T_{1wt}$ can be calculated as $(T_{RTT} - T_{pro})/2$. To mitigate the network interference, we sample 100 packet records and chose the minimum one as the one way transmission time. Then the clock drift $\Delta T_{skew}$ between the master node and monitoring node can be treated as $|T_1 + T_{1wt} - T_2|$ and this value is used for tracing data offline calculation and analysis.

### 4.3.3 Efficiency

As the position of tracepoints, rules and actions are defined by users through configuration files, the tracing scripts are normally attached to those tracepoints and execute the corresponding actions when monitored events happen. Figure 4.5 illustrates the workflow of tracing on a network device. When a packet goes across a network interface, the original process is just to pass it to the next layer or network device. However, when an tracing script is attached to the interface, the program will be executed and check whether the packet matches the user defined rules. If it matches, the user-defined actions, such as recording the system time, updating the counters, etc., are executed in the tracing script. Afterward the

---

*Figure 4.5: How eBPF code works for packet filtering and tracing.*
raw tracing data is copied into the local memory associated with this tracing script. If the above actions finish or the rules are not matched, normal packet processing proceeds.

As the network monitoring might generate lots of intermediate tracing data, the overhead of vNetTracer will be extremely high if the storage of that temporary tracing data involves too many disk operations like traditional logs. Such overhead is unacceptable and might hurt the application performance, especially for high speed network services or highly consolidated virtualized networks. To mitigate this issue, we load a kernel module on each monitoring machine to temporarily store the intermediate tracing data. We used `mmap()` to map a kernel buffer to the `/proc` file system in user space. When the tracing scripts generate some intermediate data, it is first copied to the memory buffer. Then we periodically dump the tracing data from the buffer onto the disk, clear the buffer and then collect the raw tracing data on each monitoring machine to a centralized data processing node. As we can adjust the memory buffer size\(^1\) to make the data be stored and collected infrequently, the above steps will not incur so much overhead as to affect the application performance in virtualized networks.

The tracing data is collected by the raw data collector from monitoring nodes to the master node. The collection can be processed either online or offline. For applications which require realtime monitoring, tracing data could be sent from agents to the collector directly. However, such a process could consume additional CPU and network bandwidth. For applications whose QoS is sensitive to the network or tracing overhead, the tracing data can be collected offline. After tracing is completed, all the tracing records at different tracepoints are dumped into the trace database, where records are indexed by their packet IDs. After the data cleaning and recomputation, such as identifying incomplete records, timestamp alignment for the clock skew, etc., one then can query the database to perform customized analysis of network performance. As the raw data collector periodically receives tracing data from the agents, it also acts as a heartbeat monitor to guarantee that the agents work properly.

\(^1\)Due to the limitation of `malloc` in Linux, the buffer size range is from 32 bytes to 128k-16 bytes.
4.3.4 Programability

After the agents trace the network activities and the data is collected by the raw data collector, additional calculation is required based on those raw tracing data. In this section, we briefly introduce the network performance metrics that vNetTracer is focusing on.

Throughput. The network throughput measures the amount of network packets transmitted from one side to another during a certain period of time. To quantify the throughput at a specific place, e.g., one network socket interface or a kernel function, we track the packet size $S_i$ and the arrival time $T_i$ during the data transmission, and calculated the network throughput as $\sum_{i=1}^{N}(S_i - S_{ID})/(T_N - T_1)$, where the $i$ refers to the order of the network packets during the transmission and $S_{ID}$ is the 4 bytes packet unique ID.

Latency. The network latency measures the time that one packet is transferred from one designated point to another. Based on the packet ID mentioned in Section 4.3.2, we track two packets for the same packet ID at two tracepoints and record the system time through tracing scripts. Suppose the time we record at the two tracepoints are $t_1$ and $t_2$. If these two tracepoints are within the same monitoring node, the latency between the two tracepoints is treated as $\Delta T = t_2 - t_1$. If the two tracepoints are located on two different nodes, the latency can be calculated as $\Delta T = t_2 - t_1 + \Delta T_{skew}$.

Jitter. Besides the absolute latency value, the variability of packet latency over a period of time, named jitter, is also important, especially for realtime applications such as video services and live broadcast. Based on the latency measurement, we calculate the network jitter as $\Delta T_{i+1} - \Delta T_i$, where the $\Delta T_i$ refers to the $i$th network latency of traced packet.

Packet loss. Packet loss occurs when the packets are transferred across the network but fail to reach their destination. Packet loss is usually caused by network congestion, network disconnection, device failure, etc. To measure packet loss, we track the number of packet $N_i$ at each tracepoint and calculate the packet loss between two tracepoints as $N_{loss} = N_i - N_j$ and the packet loss rate as $R_{loss} = N_{loss}/N_i$.

Additional metrics. Beside the above basic metrics, more information could also be dug from the raw data for certain scenarios, such as packet arrival time. In addition, combined with the tracing rules, advanced tracing information, like per-flow throughput and the decomposition of end-to-end latency, which are illustrated in Figure 4.6, can be obtained based on the user needs.

The programmability of vNetTracer also allows the user to control the tracing at runtime. Unlike traditional tools which couple the monitoring logic with system execution or need to stop the system for new tracing logic, vNetTracer strips the tracing from the monitored system. We encapsulate the network tracing into highly configured eBPF scripts and execute them at certain tracepoints based on monitoring purposes. As shown in Figure 4.2, we separate the vNetTracer control plane from the application network flow and tracing.
data flow. To realize that, we created highly modularized control package, which includes the tracing rules, tracepoint locations, actions and global configurations, for each tracing script. During the execution, the vNetTracer control data dispatcher formatted the user requirements into tracing configuration files. For instance, users provide information such as ethernet type, source IP, destination port, etc. to generate the filter rules, or file names, function, device ID, etc. to generate tracepoint locations. Once the tracing configuration files are defined, customized tracing scripts are sent to tracing agents across the system and collect the data. When the networks or tracing requirements change, all the above control information can be modified or reconfigured, and then resent to the monitoring nodes during the system runtime. Such a process provides high programmability and flexibility for monitoring the dynamic virtualized networks.

4.3.5 Implementation

To enable tracing across boundaries, we added 4-byte variable option in tcp_out_options for the TCP header and allocated 4-byte space for the UDP header. The unique ID is written when packets are sent through tcp_options_write or udp_send_skb. To mitigate the local storage overhead, we implemented a kernel module on each monitoring node to temporarily store tracing data and used /proc file system to avoid kernel and user space data copies. The prototype of vNetTracer is implemented in C and Python. Specifically, the agents are daemon processes, which are woken up once receiving new tracing scripts. The backend of each tracing script is implemented in C, which executes actions inside the kernel and collects the tracing data. The frontend is implemented in Python, which completes the initialization, stores local data, and periodically sends the tracing data to the collector. The control data dispatcher consists of a frontend, which reads the user input from terminal and generates the formatted configuration files, and a client side that sends the configured tracing scripts to the remote agents. The raw data collector consists of a daemon, which receives tracing data from the agents, and the database operation functions to calculate the network metrics. Both the control data dispatcher and the raw data collector are implemented in Python. We adopt InfluxDB for the offline storage and create tables for each tracepoint.

4.4 Evaluation

In this section, we first evaluate the overhead of vNetTracer to show its high performance. Next, we demonstrate the utility of vNetTracer with three case studies. The first case study is to show critical path analysis in Open vSwitch with vNetTracer. The second case study is using vNetTracer to tune the Xen hypervisor scheduler for a long tail latency
Figure 4.7: vNetTracer incurs negligible overhead to (a) application average latency, tail latency, and (b) throughput. Compared to SystemTap, the overhead is marginal due to the eBPF benefits and the optimizations to vNetTracer.

issue in a highly consolidated environment. The third case study shows how we identify the network bottlenecks inside a container architecture using vNetTracer.

### 4.4.1 Evaluation Settings

Our experiments were performed on two DELL PowerEdge T430 servers, connected by a one-Gigabit Ethernet and a ten-Gigabit Ethernet. Each server was equipped with a dual ten-core Intel Xeon E5-2640 2.6GHz processor, 64GB memory and a 2TB 7200RPM SATA hard disk. The ten-Gigabit NIC is Intel x540. We used Ubuntu 16.10 and Linux kernel 4.10 as the host and guest OS. We used Open vSwitch 2.6.0 to connect various VMs on the same host. The hypervisor we adopted is KVM 2.6.1 or Xen 4.8.1, and the Docker version is 1.12.1. The evaluation setting details of individual case studies are slightly different and further discussed in the respective sections.

### 4.4.2 Overhead Analysis.

**Overall overhead.** We first analyzed the overhead of vNetTracer on application network performance. We created two VMs using KVM on two servers and configured each VM with 4 vCPUs and 4GB memory. We pinned the vCPU of the VMs to different physical CPU cores to avoid the interference. First, we executed `Sockperf` [28] client side on one VM and sent UDP requests to the Sockperf server side on another VM to measure the average and tail latency. Then we booted vNetTracer to trace the Sockperf performance. We executed four tracing scripts and attached them into the Open vSwitch port `ovs-br1` in the hypervisor and virtual ethernet port `ens3` in the VM on the two physical servers. Figure 4.7(a) plots the average and 99.9\(^{th}\) percentile latency of Sockperf UDP packets with and without vNetTracer execution. As shown in the Figure 4.7(a), both average and tail latency of Sockperf were not influenced significantly with vNetTracer. Compared to the
default performance without vNetTracer, the average latency with vNetTracer increased less than 1%, and no traffic burst happened during the tail latency measurement. Meanwhile, our tracing also showed that vNetTracer did not introduce additional network packet loss for the applications.

**Comparison with SystemTap.** We also compared the performance of vNetTracer with SystemTap. We built a VM which had one vCPU and 4GB memory on Xen and executed the Netperf server inside the VM. A Netperf client was sending TCP packets on another physical server. We wrote a SystemTap script attached at `tcp_recvmsg` to get the network packet. We executed the SystemTap with option `STP_NO_OVERLOAD` to disable the tracing overhead threshold. In comparison, we used vNetTracer to attach the same kernel function and trace the network. As shown in Figure 4.7(b), due to the benefit of eBPF and our optimizations, the throughput of Netperf degraded insignificantly when tracing under vNetTracer. However, SystemTap tracing introduced around 10% performance loss. As explained in Section 4.2, the frequency of traces and the continual data copies between the kernel space and user space introduced such overhead with SystemTap. vNetTracer traces the network inside the kernel and keeps the tracing data in memory. For I/O-bound applications with high load, such the overhead cannot be neglected. We also evaluated the Netperf performance on a 10G network and SystemTap introduced 26.5% performance loss due to high frequency of traces and inefficient data copies.

### 4.4.3 Case Study I: Network Delay in the Open vSwitch

Open vSwitch (OVS) is a virtualized network switch, which provides high quality packet switching for virtualized networks and is widely adopted in the current cloud. In this section, we first describe the network delay inside the OVS and discuss the challenges to diagnose the problem. Next, we talk about how vNetTracer helps us analyze and locate
the issue. Lastly, based on the tracing information, we share a simple yet effective solution to mitigate the above issue of Open vSwitch.

In order to describe the network delay inside the OVS, we created three VMs on a single physical server. The hypervisor we used was KVM and all the VMs were connected through OVS. The VMs were configured with four vCPUs and 4GB memory. As shown in Figure 4.8(a), we executed the Sockperf and iPerf [16] clients on VM0, another iPerf client on VM1, and the Sockperf server as well as two iPerf servers on VM2. As a comparison baseline, we only run the Sockperf application to measure the latency in an uncongested network, which is denoted as Case I in Figure 4.8(b). Next, we run the iPerf client with Sockperf client simultaneously on the same VM and record the Sockperf latency as Case II. Finally, we add the second iPerf client on another VM based on Case II and denote such a scenario as Case III.

As illustrated in Figure 4.8(b), the tail latency of Sockperf in Case II and Case III increased significantly compared to the latency in the uncongested network. Similar problems can also be observed in the physical switches [73, 164]. The reasons behind the issues are extremely difficult to analyze and locate as many factors along the data path can introduce the additional overhead. For example, the latency-intensive and throughput-intensive applications in Case II share both the client and the server network stack, which might incur long tail latency for Sockperf. The existing tracing techniques either focus on a traditional environment, such as physical network links, or lack of ability to differentiate network flows and locate the congested part. To analyze the bottlenecks in the virtualized network path, we executed agents of vNetTracer on both the VMs and host machine, and bound the tracing scripts at application sockets \texttt{em} and OVS ports \texttt{vnet}. We used eBPF scripts to filter the Sockperf packet and decompose its latency into three parts: the time spent inside the sender network stack, the OVS and the receiver network stack. As the latency decomposition shown in Figure 4.9(a), the time spent inside the OVS dominated
the total transmission time. As more applications occupied the network path, the network became increasingly congested and the time in the OVS increased.

To better understand the network delay inside the OVS, we add more iPerf clients on VM0 based on Case II as Case II+. Similarly, we add more iPerf clients on additional VMs and denote that as Case III+. As shown in Figure 4.9(a), the time gap between Case I and Case II is due to the queueing delay in the OVS. Multiple applications (e.g., Sockperf and iPerf) send network packets at the same ingress port of OVS and the delivery speed of OVS falls far behind the packet incoming speed. Such a gap does not increase when we added more the application clients on VM0 in Case II+ because the queue at ingress is highly saturated. In comparison, the time gap between Case II and Case III, which results from the processing delay that OVS needs to switch the network flows from different ingress ports, increased when more clients are sending packets through more OVS ingress ports in Case III+. Both the above two delays make the network flow load much larger than the OVS processing ability and introduce additional network latency inside OVS. Compared to the delay inside OVS, the time spent inside the client or server network stack did not increase significantly.

To mitigate the above issues, one potential solution is to limit the network flow rate at the OVS ingress. Rate limiting sets the maximum packet transmission rate at the virtual ingress port of OVS, which just simply drops the network packets above the rate and can thus limit the packet transmission number from VMs to the OVS. In our experiment, we set the ingress_policing_rate as $1 \times 10^5$kbps and ingress_policing_burst as $1 \times 10^4$kb at both vnet0 and vnet1 of the OVS. Then we repeated the above experiments in Case II and Case III. As shown in Figure 4.9(b), both the average and tail latency of Sockperf decreased significantly with rate limit in the OVS. As the default Sockperf packet size was just 56 bytes, the workflow of iPerf was mainly limited when its packets entered the OVS. Therefore, both the queueing delay and OVS processing delay were mitigated with the rate limit. The setting in the above experiments was not the optimal configuration and just used to show the effectiveness. In addition to the rate limit, we also tried setting QoS policy with Hierarchy Token Bucket (HTB) at the virtual port of OVS, which limited the clients saturating the network bandwidth. The effect was similar as the results using rate limit shown in Figure 4.9(b).

Summary. Existing tools either lack of ability to differentiate the network with complex flows or cannot decompose the long tail latency into different components along the data path. In comparison, vNetTracer can filter and monitor the target network flow, and locate the potential congested component in the virtualized networks efficiently.

39
4.4.4 Case Study II: Tuning the Scheduler in Hypervisors

Credit2 [38] is a new generation of general purpose scheduler for the hypervisor Xen, which is designed with focus on fairness, responsiveness as well as scalability. In this section, we first describe the issue we found in the current credit2 scheduler. Then, we talk how we located the issue with vNetTracer and solved it through tuning the scheduler.

We created two VMs on a single physical server. The hypervisor we used was Xen 4.8.1 and the VMs were configured with one vCPU and 4GB memory. The hypervisor scheduler was set as credit2 inside of Xen and the client side was executed on another physical server. All the applications were running within containers on the VMs. First, we executed the Sockperf server side on one VM and sent requests to measure the latency as the baseline. Next, we executed a loop on another VM and pinned the vCPU of the two VMs on the same physical CPU core. As shown in Figure 4.10(a), the latency of Sockperf increased dramatically when the I/O-bound VM shared the CPU resources with the CPU-bound VM. For instance, the 99.9\textsuperscript{th} percentile latency increased 22x compared to the baseline. Besides, we also chose Data Caching from Cloudsuite benchmark suites [5] to evaluate this issue. The server side of Data Caching executed Memcached which simulated the behavior of a Twitter caching server using the Twitter dataset. On the client side, we set up 4 worker threads executing 20 connections to send the requests and the ratio of GET/SET requests was configured as 4:1. We set a fixed request rate as 5000 rps to measure the request latency. As depicted in Figure 4.10(b), similar to the results of Sockperf, the average and tail latency of memcached increased 4.7x and 7.5x respectively compared to the baseline. The greatest challenge to analyze this issue is the multi-layer virtualization, including the hypervisor, the guest OS and the containerized applications, and the complicated virtualized network along the software stack. These virtualized boundaries make many traditional tools, such as Xentrace [33], DTrace [9], ineffective.

Figure 4.10: Network latency in the unconsolidated and highly consolidated virtualized environments.
In order to analyze this problem, we executed the agents of vNetTracer at the client, Dom0 and server side VM to trace the network. We bound the tracing scripts at the following network interfaces: ethernet port eth0 in the client side, network bridge xenbr0 and backend vif1.0 in Dom0, ethernet port eth1 and container virtual ethernet port veth684a1d9 in the server VM. We decomposed the packet latency based on the above setting and repeated the experiments. As shown in Figure 4.11(a), when the I/O-bound VM executed alone, the client-to-server transmission delay dominated the one way latency. In comparison, when the I/O-bound VM shared the same CPU core with the CPU-bound VM, the time spent between the backend vif1.0 in Dom0 and frontend eth1 in the server VM took more than 90% of the one way latency. As the data path was the same as in Figure 4.11(a), this indicated that scheduling delays inside of Xen caused the issue. We checked the source code in Xen credit2 scheduler and found the vCPU priorities used in credit1, such as OVER, UNDER and BOOST, were all removed and all the vCPUs were just ordered by their credit. Then we traced vCPU credit and the data showed that the credit of the I/O-bound VM vCPU was always larger than the credit of the CPU-bound VM vCPU, which indicated that the scheduling order of vCPUs had no problem. We further analyzed the tracing data and found that the scheduling delay first increased up to 1000 $\mu$s. Then the scheduling delay descended for the next few packets and such process repeated periodically as shown in Figure 4.11(b). That reminded us of the scheduling rate limit inside the Xen credit2 scheduler, which is set as 1000$\mu$s by default.

The rate limit was introduced into the hypervisor credit scheduler since Xen 4.2 [37]. In order to avoid too many scheduleings and context switches, the scheduler sets the minimum amount of time which a VM is allowed to run without being preempted, even through a woken VM has higher priority. This mechanism performs well and does not harm the

---

2 The rate limit in Case Study I refers to limiting the application packet sending rate while the rate limit here refers to the minimum scheduling time slice inside the Xen hypervisor.
throughput of most network applications. However, the average latency as well as the tail latency of many online applications is highly interfered with such a mechanism, especially in the highly consolidated virtualized environments. In addition, the jitter of the I/O application also increased significantly. For instance, the range of jitter in Figure 4.11(a) was only (-7.2 μs, 9.2 μs) while the value grew to (-117.8 μs, 1041.4 μs) in Figure 4.11(b). To mitigate such issues, we tried to tune the rate limit as 0 in Xen credit2 scheduler. As shown in Figure 4.10(a) and (b), the network latency with rate limit disabled is close to the baseline even though the I/O-bound VM runs simultaneously with the CPU-bound VM on the same physical core. Such a solution also works for the same issue in credit1 scheduler inside Xen. We reported our findings to the Xen open source community and the above issues were confirmed by engineers from Citrix [20].

Summary. Unlike many tools limited in monitoring within certain ranges, vNetTracer can efficiently trace the application end-to-end performance in the virtualized networks and associate the network activities across hardware and software boundaries in isolated domains.

4.4.5 Case Study III: Bottlenecks of the Container Architecture

Containers are widely used in the cloud nowadays and the overlay network is one of the extensively adopted infrastructures to support the container communication across multiple hosts. Although overlay networks bring lots of benefits, such as ease of use, independence from the underlying architecture, etc., they also introduce many new issues. For instance, the performance of the overlay network is usually much worse than the other kinds of container networks [143]. Moreover, it is also difficult to monitor the overlay network as the packets are encapsulated with the underlying network information. In this section, we describe how vNetTracer helps us analyze the bottleneck in a container overlay network.

As depicted in Figure 4.12(a), we created two VMs using KVM on a single physical server. Each VM was configured with 4 vCPUs, 4GB memory and a virtio NIC. Inside the VMs, we used Docker to create some containers executing the network applications. In order to connect the containers on the two VMs, we built a default Docker overlay network and used etcd [10] 2.2.5 as the distributed key value store. The overlay network used VXLAN to encapsulate the original network packets. We used Netperf and iPerf to measure the throughput among VMs or containers. As depicted in Figure 4.12(b), the TCP throughput between containers decreased significantly compared to the VM throughput. For instance, the Netperf TCP and UDP throughput between containers were just 16.8% and 22.9% of that between VMs, which indicated significant overhead in the container overlay network.
Figure 4.12: Container multi-host networking setting and the Netperf/iPerf throughput of VMs versus containers.

To understand the potential issue inside the container overlay network, we first analyzed the network rate in the virtualized network stack. We attached tracing scripts on the kernel function `net_rx_action`, which is the default `softirq` handlers when receiving network packets. As Figure 4.13(a) shows, although the throughput of containers is far less than that of VMs, the execution rate of `net_rx_action` in containers is 4.54 times of that in VMs. This indicated that more `softirqs` happened when receiving network packet in the container network. Why do additional interrupts introduce such significant overhead? First, additional interrupts incur lots of context switches. As revealed by Peter et al. [122], the scheduling overhead differs by up to 14x difference depending on whether the receiving process is currently running. The time to context-switch to the server process from the idle process has more than 10x impact on a receiving process. Second, too many interrupts might cause lots of sleep and wakeup operations to the `ksoftirqd`, which is a daemon thread that executes on each CPU core to handle the software IRQs. The interrupt processing of `ksoftirqd` will be significantly interfered if too many sleeps and wakeups consume lots of CPU cycles [91].

Besides the network rate, we also used vNetTracer to analyze the `softirq` distribution and track packet data path inside container networks. We attached tracing script at the kernel function `get_rps_cpu` to get which CPU core the `softirq` is processed on. As Figure 4.13(a) shows, most `softirqs` are concentrated to a few CPU cores. For instance, 99.7% and 62.9% of the `net_rx_action` is executed on CPU 0 in VMs and containers, respectively. However, as mentioned above, the number of `softirqs` in containers is far more than that in VMs. Therefore, such concentrated `softirq` distribution and lack of multi-core usage heavily limited the network speed in containers. The reason is due to the fact that those software interrupts come from the same hardware interrupt on the NIC. To keep the cache hot and take full advantage of data locality, OSes process the `softirqs` from the same source on certain cores, which significantly degrades container network performance with
large number of software interrupts. Receive Packet Steering (RPS), which helps balance network packets on different CPU queues, is also limited for accelerating the container networks. As RPS only balances the packets based on its IP addresses and port numbers and packets of one containerized application connection has the same IP and port combination, all the software interrupts of one application are processed in the same CPU and the container network performance will not benefit with RPS enabled.

To analyze the packet data path, we attached multiple eBPF scripts at different layers in the virtualized network stack to track the source and destination of the network packets. The device ID and name are acquired by the `ifindex` and `name` in `net_device` of the skb. Note that the tracing scripts need to strip the VXLAN header off to read the skb information in container networks. As shown in Figure 4.13(b), the data path in container networks is far more complex than that in VMs, which also implicated the enormous amount of software interrupts in Figure 4.13(a). Different from the normal packet processing inside VMs, the packets travel across different layers repeatedly in the container networks. This is due to the overlay network architecture which abstracts additional virtualized network layers on top of the VM network. During traveling in such much deeper and more complex container networks, additional efforts, such as security checks, header operation, network forwarding, etc., are needed for the packets [122]. All efforts consume additional resources and slow down the network processing. In addition, vNetTracer also revealed many other details such as the bottlenecks at the Docker bridge `docker0`, the latency at the VXLAN device, the additional acknowledgment overhead for TCP transmission, etc. Although optimizing container networks is a new and complex topic, the valuable statistics provided by vNetTracer can still reveal the potential reasons behind the poor container network performance in Figure 4.12(b) and shed light on solving this problem in our future work.

**Summary.** Characterizing the Linux kernel, especially for analyzing network performance, is challenging. With vNetTracer, we can attach user-defined eBPF programs into
the systems and instrument the runtime environment in a highly efficient and customized manner.

4.5 Conclusion

It is important to trace network performance in order to guarantee application performance and analyze potential issues. However, virtualized networks make that increasingly challenging as they introduce additional complex infrastructure and changes dynamically. This work presents vNetTracer, a highly efficient and programmable profiler that traces network performance in the virtualized systems. vNetTracer incurs negligible overhead to network performance and does not require any changes to the user level applications for end-to-end network tracing. In addition, it can also be programmed and configured at runtime to satisfy various tracing requirements. Our evaluation and case studies demonstrated that vNetTracer shed light on the virtualized network monitoring and can help users analyze, identify and localize potential issues inside virtualized networks.
CHAPTER 5
XBALLOON: PRESERVING I/O PRIORITIZATION IN VIRTUALIZED OSES

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

5.1 Introduction

Due to its support for multi-tenancy, virtualization is becoming ubiquitous in datacenters. Popek and Goldberg’s virtualization requirements [123] suggest that a program running in virtualized environments should exhibit a behavior essentially identical to that in physical environments. This property does not hold for many programs, especially those equipped with their own resource management, e.g., operating systems (OSes). The culprit is the semantic gap between physical and virtual environments. Virtualization presents the illusion of dedicated hardware, but resources are often multiplexed among users and have in fact discontinuous availability.

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

Unfortunately, the co-location of I/O- and compute-bound workloads in a virtual machine (VM) can cause severe degradation of I/O performance. In virtualized environments, CPU multiplexing/sharing and capping are widely adopted to improve resource utilization and performance isolation. For example, VMware suggests that a physical CPU (pCPU) can be shared by as many as 8 to 10 VMs [152]; AWS instances with capped CPU capacity, e.g., m1.small, m1.medium, t1.micro, and m3.medium, account for around 40% of Amazon EC2 usage at RightScale [3]; the new generation AWS T2 burstable performance instances use CPU cap to provide the baseline CPU performance [1]. Either CPU capping or sharing makes the CPU availability discontinuous to a VM. Existing studies have found that discontinuous CPU availability can delay I/O scheduling and affect TCP congestion control [54, 65, 153], interrupt handling [43] and latency-sensitive workloads [155, 161, 162] even the CPU allocation to a VM is well above the I/O demand. However, a key question remains unanswered – why I/O prioritization in the guest OS is not doing its job?

In this work, we discover two types of priority inversions in the guest OS when a VM runs a mixture of I/O and compute workloads and executes on discontinuous CPU.

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

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

\footnote{Only user-level I/O-bound tasks need to be identified while kernel threads and interrupts inherently have a higher priority than any userspace tasks.}
the balloon, the guest OS can autonomously decide if the VM should be paused to reserve CPU for future use.

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

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

Figure 5.1: Network I/O suffers poor and unpredictable performance in virtualized environments. (a) and (c) Virtualization alone does not necessarily degrade I/O throughput or latency under continuous time. (b) and (d) Discontinuous time leads to significant performance loss and increasing unpredictability in VMs and containers with mixed I/O and computation.
5.2 Background and Motivation

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

5.2.1 Assumptions

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

5.2.2 I/O Prioritization in Linux

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

5.2.3 Time Discontinuity

In general, there are two ways to control the CPU allocation to a VM: CPU sharing and capping. In CPU sharing, the hypervisor consolidates multiple virtual CPUs (vCPUs) on the same pCPU. VMs take turns to run on pCPUs in a weighted round robin manner. CPU capping [7] sets an upper limit on the CPU time a VM receives, e.g., T2 Instances [1] on Amazon EC2 [2]. VMs are temporarily suspended if their CPU usage exceeds the cap.
The period during which a VM is not running, either due to sharing CPU with other VMs or capping, creates gaps on VM perceived time. These gaps cannot simply be overlooked by VMs. The guest OS needs to keep up with the host wall-clock time to avoid time drift. Most OSes today para-virtualize its clock to synchronize with the hypervisor. Thus, the guest OS sees discontinuous passage of time when the VM is de-scheduled and later scheduled.

5.2.4 Degraded I/O Performance due to CPU Discontinuity

In this section, we show that Linux fails to preserve I/O prioritization when running as a virtualized OS under discontinuous time. We emulated a scenario in which a VM is loaded with heterogeneous workloads. Netperf [22] and sockperf [28] benchmarks were used to test network throughput and latency with multiple hypervisors and an OS container. The workloads were run in one-vCPU VMs in KVM [17], Xen [47] and a Docker container [8], respectively. The VMs and container ran the Linux 3.18.21 kernel and were configured to run in two modes: full and partial CPU capacity. Under full capacity, one physical CPU (pCPU) was dedicated to the VMs or the container. Thus, CPU was always available and continuous to the guest OS. Under partial capacity, we limited the VMs or container to use 50% of the pCPU. As such, the guest OS ran for a certain period and was suspended for the same amount of time to satisfy the constraint. With partial capacity, the Linux guest experienced discontinuous CPU availability.

A while(1) CPU hog was used to emulate a background compute-bound task that co-ran with the I/O task in the guest OS (denoted as I/O+CPU). The reference I/O performance was obtained when the CPU hog was turned off (denoted as I/O only). Since I/O-bound tasks consume less CPU, Linux gives them higher priority than compute-bound tasks under CFS. Figure 5.1 (a) and (c) show that Linux effectively preserves I/O prioritization under continuous CPU. The CPU hog did not degrade network throughput and even improved tail latency. The co-location of I/O- and compute-bound tasks prevented guest OS from entering low-power states (i.e., C-states), reducing the startup cost to service network requests.

In contrast, both hypervisors and the container suffered significant throughput loss and latency hike under discontinuous CPU. While the reference I/O performance dropped due to reduced CPU capacity, the compute task caused up to 65% further throughput loss (i.e., Xen) and 100x latency increase with 27% variation (i.e., Xen) in 10 runs (as shown in Figure 5.1(b) and (d)). Network latency in Figure 5.1(d) in the I/O only case was in microseconds, thereby not showing up in the figure. These experiments clearly show that Linux guest’s property of prioritizing I/O operations over compute activities is not preserved when the VMs and container experience discontinuous CPU availability.
5.3 Analyzing Priority Inversions

5.3.1 Short-term Priority Inversion

Linux’s CFS uses vruntime to track task CPU usage and prioritizes those with small vruntimes. Short-term priority inversion happens when the vruntimes of I/O tasks are mistakenly dilated under discontinuous time so that CFS fails to prioritize them. Figure 5.2 illustrates how vruntime dilation can happen under discontinuous time. If the VM is suspended right after a process starts running on a vCPU, the period in which the VM is not running will be charged to the process because the VM synchronizes its clock with the host after resuming execution. Thus, vruntime update of the process will include the time gap. Inaccurate time accounting does not affect tasks with static priority as task scheduling is not based on CPU usage. In contrast, time dilation can interfere with vruntime-based scheduling in Linux CFS.

Figure 5.3 illustrates how time dilation affects the scheduling of an I/O-bound task but not a compute-bound task. We co-located a sockperf server process with a CPU hog in a one-vCPU VM that is capped at 50% capacity of a pCPU. Figure 5.3 plots the vruntimes of the two tasks whenever they were scheduled or descheduled by CFS. The number of dots reflects how long a task runs on CPU. After the first time gap, the vruntime of the I/O task was dilated and it became much larger than the vruntime of the compute task. As a result, the I/O task was not scheduled by CFS until the compute task’s vruntime caught up with that of the I/O task. In comparison, after the second time gap, even though the compute-bound task suffered time dilation, it was still scheduled by CFS in a fair manner. This is because CFS clamps a waking I/O task’s vruntime to the minimum vruntime in the runqueue minus two time slices [96]. Therefore, the I/O task’s adjusted vruntime also included the time gap. As a consequence, inaccurate time accounting only penalizes I/O tasks in CFS scheduling.
Figure 5.3: Time dilation only affects the scheduling of I/O-bound tasks in Linux CFS.

5.3.2 Long-term Priority Inversion

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

Figure 5.4(a)-(e) show how long-term priority inversions develop between a high priority task (e.g., I/O-bound task, white bar) and a low priority task (e.g., compute-bound task, grey bar) under discontinuous time due to a 50% CPU cap. The higher priority of the I/O task can be either statically assigned by the administrator or dynamically determined by CPU usage. Figure 5.4(a) shows the CPU allocations between the two tasks under continuous time. The I/O-bound task consumes less CPU than the compute task thus can always preempt the latter when waking up. Note that Figure 5.4(a) demonstrates the overall CPU utilizations of the two tasks during a period of time, in which many individual I/O requests are serviced (as shown in Figure 5.4(e)). Due to a lower priority, the compute task only runs when the I/O task is idle.

Work-conserving scheduling makes sense under continuous time but can violate I/O prioritization under discontinuous time. As shown in Figure 5.4(b), neither static nor dynamic priority can be preserved with the 50% CPU cap. For example, if static priority had been enforced, the CPU allocation to the high priority I/O task would not be affected (as shown in Figure 5.4(c)) and only the low priority CPU-bound task should receive reduced allocation. With dynamic priorities, both the demands of the I/O (i.e., 30% demand) and the compute tasks (i.e., 70% demand) exceed the fair share (i.e., 25%) under the new CPU
Figure 5.4: Work-conserving scheduling in guest OS does not preserve priorities between a high priority (white bar) and a low priority task (grey bar) under discontinuous time.

Running Linux in a VM or container with constrained CPU allocation violates both types of priorities. Figure 5.4(b) shows the CPU allocation between the two tasks due to WC scheduling. The CPU allocation to the high priority I/O task is significantly lower than those in Figure 5.4(c) and (d). The problem is that the guest OS assumes dedicated and continuous CPU resources but actually runs on shared, discontinuous CPU allocations. Thus, WC scheduling allows the compute task to consume the CPU that could otherwise be used by the I/O task in the future. Unlike in dedicated systems, where I/O tasks can always timely preempt compute tasks, I/O tasks in a VM with discontinuous CPU are unable to acquire CPU if the VM are not scheduled. As illustrated in Figure 5.4(e), the period, during which the VM is suspended (dotted bar) due to the cap, prevents half of the I/O requests from being processed. The delay can significantly degrade I/O performance, especially the tail latency shown in Figure 5.1(d).

### 5.4 xBALLOON Design

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

KVM implements two clocks in Linux guests to address inaccurate CPU accounting due to discontinuous time [18]. While rq_clock synchronizes with the host clock, rq_clock_task only ticks when a VM is running. CFS schedules tasks based on rq_clock_task so that vruntimes truly reflect task runtimes. Thus, the short-term priority inversion problem has been addressed by KVM. We port the relative clock in KVM to Xen VMs as rq_clock_virt and make it available to other Linux schedulers. Besides preventing short-term priority inversion, the two differential clocks also help enforce static priority between tasks. Note that as shown in Figure 5.4(c), the low priority task should be the victim of reduced CPU allocation if static priorities are enforced. As will be discussed in § 5.4.4, the absolute clock rq_clock is assigned to the low priority task so that the deprived CPU is accounted to its consumption.

5.4.2 CPU Balloon

The idea of CPU balloon is inspired by memory ballooning [154], in which the guest OS installs a balloon driver for dynamic memory allocation. The size of the memory balloon represents the amount of memory that has been taken away from the guest. Thus, dynamic memory allocation is realized by inflating and deflating the balloon. Similarly, we use a CPU balloon to represent CPU time the VM voluntarily gives up and reserves for future use.

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

The purpose of the balloon process is to prevent low priority compute tasks from running when the high priority I/O task is idle, which effectively converts the original work-conserving scheduling to non-work-conserving. However, the balloon should only run under constrained CPU allocation and be disabled if there is CPU resource slack to allow the compute task to run freely. Next, we present the resulting semi-work-conserving scheduling (§ 5.4.3) and show how to precisely preserve static (§ 5.4.4) and dynamic (§ 5.4.5) priorities.

5.4.3 Semi-Work-Conserving Scheduling

The goal of semi-work-conserving (SWC) scheduling is to differentiate task scheduling based on the availability of CPU. Under constrained CPU allocation, the VM would be
forcibly suspended or de-scheduled if its CPU consumption exceeds the CPU cap or the fair share. To preserve priorities, the guest OS should be in total control of its CPU usage and avoids involuntary suspension and descheduling by the hypervisor, while still meeting the resource constraint. As illustrated in Figure 5.5, to avoid suspension during the black-out period (dotted bar in Figure 5.5(b)), the VM autonomously schedules the balloon process to reserve CPU for I/O processing. To enforce static priority, as shown in Figure 5.5(c), the compute task only runs when all I/O requests are serviced. To preserve dynamic priority, as shown in Figure 5.5(d), the execution of the compute task is interleaved with that of the I/O task. Ideally, the execution time of the balloon, i.e., the reservation of CPU, equals the length of period in which CPU is unavailable to the VM. If so, the VM proactively satisfies the resource constraint and avoids involuntary suspension.

Figure 5.6 shows how the SWC scheduling realizes such autonomy. During the NWC mode, the balloon is active and throttles the execution of the compute task. The guest OS scheduling switches back to WC mode when the balloon is suspended. The challenge is how to switch between the two modes so that the demand of the I/O task is fully satisfied and the compute task is free to run if there exists CPU slack. Recall that WC scheduling does not violate I/O prioritization under continuous time because the I/O task is always able to preempt the compute task. If we can preserve this property, I/O performance would not be affected by the compute task even under discontinuous time.

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

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

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

**SWC workflow.** As shown in Figure 5.6, at the beginning of each accounting period, the guest OS is in NWC mode and the balloon process is active (step 1). Each time the balloon is scheduled to run, it calls down to the Linux kernel and checks the current mode from a per-CPU variable xballoon_mode shared with the hypervisor. The mode switches to WC once the hypervisor finds that the maximum amount of credits can be debited from
the VM until the next credit refill will not cause either a VM suspension due to capping or an entry to the OVER state. If so, the balloon suspends itself and waits on a task sleep queue balloon_queue (step 2). When the next accounting period starts, the hypervisor notifies the guest OS to wake up the balloon and switch to NWC (step 3).

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

5.4.4 Enforcing Static Priority

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

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

5.4.5 Preserving Dynamic Priority

The dynamic priority between the I/O and compute tasks can change depending on the demands of individual tasks and the fair share under the constrained CPU allocation. For example, as shown in Figure 5.7(a), the I/O task, which has a high priority due to
smaller runtimes under continuous time, should receive an equal priority as the compute task and a fair share of CPU under the constrained allocation. In contrast, if the I/O task still demands less than the fair share (shown in Figure 5.7(b)), its demand should be fully satisfied and it is assigned a higher priority.

In Linux CFS, dynamic priorities are determined by vruntimes. As illustrated in Figure 5.5(a) and (b), I/O requests (white bars) arrive at discrete time and the real I/O demand, i.e., the aggregate of all white bars under continuous time, cannot be fully presented to CFS for scheduling under discontinuous time, thereby violating the dynamic priority. To address this issue, we integrate the balloon process into CFS scheduling and extend the CFS fair sharing to include CPU reservations for future use (i.e., the balloon). As shown in Figure 5.5(c), if the balloon (the black bar) is properly scheduled, the I/O demand can be fully exposed to CFS. Next, we discuss how the balloon helps preserve dynamic priority for the two cases shown in Figure 5.7 using extended fair sharing.

**Fully satisfying I/O demand.** In extended fair sharing, the balloon, the I/O and compute tasks are scheduled by CFS as regular processes, i.e., with the SCHED_OTHER policy and all use the relative clock rq_clock_virt. Note that the balloon’s runtime also includes the time the VM is paused by the balloon. As discussed in § 5.4.2 the demand of the balloon ($r_{bal}$) is the inverse of the I/O demand ($r_{io} = 1 - r_{bal}$). Assume there are $n$ processes, including the I/O task, sharing the CPU and the VM CPU allocation is $c$. The fair share of the CPU allocation among $n$ tasks is $\frac{c}{n}$. For compute-bound tasks, their demand/runtime ($r_{comp}$) will be bounded by the fair share ($\frac{c}{n}$). If the I/O tasks demands less than the fair share, as shown in Figure 5.7(b), $r_{io} < \frac{c}{n}$ and given that $c \leq 1$, we have the following strict order between task runtimes if they are scheduled by CFS: $r_{bal} > r_{comp} > r_{io}$. Thus, the I/O task is guaranteed to have the smallest vruntime and its demand will be fully satisfied.

---

3 We assume the total CPU capacity to be 1 and the demands and allocation are in percentage.
Figure 5.8: xBALLOON improves I/O performance by enforcing static priority. The x-axis shows CPU caps.

**Enforcing fair sharing.** When the I/O demand is larger than the fair share (as shown in Figure 5.7(a)), a tweak is needed to guarantee that CFS fairly allocates CPU to the I/O task. CFS clamps a waking I/O task’s vruntime to the minimum vruntime in the runqueue minus an offset to prevent I/O tasks from monopolizing CPU. This design works effectively and correctly under continuous time as I/O tasks that demand more than the fair share are guaranteed not to have too small vruntimes compared to the runqueue minimum. However, this property does not hold under discontinuous time, where the demand of the I/O task exceeds the fair share because the fair share itself drops. Since CFS is not aware of time discontinuity, it frequently resets the vruntime of the I/O task but not that of the compute task, making it impossible for the I/O task to achieve the fair share. To this end, we make a minimal change to CFS: when the balloon is running and CFS experiences discontinuous time, the I/O task is allowed to preserve its vruntime when waking up.
5.5 Implementation

We implemented xBALLOON in Linux 3.18.21 and Xen 4.5.0. To support two differential clocks in guests, we ported the `rq_clock_task` in KVM guest to Xen VM. To support VM pause, we added a new system call (`sched_balloon`) in Linux kernel and a new hypercall (`SCHEDOP_sleep`) in Xen. xBALLOON used the existing VM pause interface in Xen, but unlike the existing `vcpu_pause` holding a per-vCPU lock, we implemented a fast path `vcpu_fast_pause` for handling frequent VM sleeps. Since VM pause is to reserve CPU for future I/O, it should be canceled if an I/O request is already pending to avoid delayed I/O processing. We added this optimization in both Xen and Linux to avoid problems such as transmission delay of TCP ACKs to senders [161].

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

5.6 Evaluation

In this section, we evaluate xBALLOON using both micro-benchmarks and real-world applications. We first created a controlled environment, where a one-vCPU VM was used and CPU discontinuity was due to CPU capping, to study the effectiveness of xBALLOON in improving I/O performance and preserving priorities (§ 5.6.1). We then show results on the CPU sharing case (§ 5.6.2) and evaluate xBALLOON with two 4-vCPU VMs running realistic workloads (§ 5.6.3). Finally, we discuss xBALLOON’s overhead and its applicability to different workloads (§ 5.6.4), and present a preliminary study on Amazon EC2 (§ 5.6.5).

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

5.6.1 CPU Capping

Capping provides a convenient means to control resource allocation and evaluate the effectiveness of xBALLOON at various levels of CPU allocation.
Figure 5.9: xBALLOON improves I/O performance by preserving dynamic priority in CFS. The x-axis shows CPU caps.

5.6.1.1 Improving I/O Performance

We begin with micro benchmarks netperf, sockperf and a synthetic disk benchmark that sequentially accesses a 2GB file to measure TCP/UDP throughput, network tail latency and disk read throughput, respectively. The compute task was a while(1) CPU hog, which had almost zero memory footprint. Figure 5.8 and Figure 5.9 show the performance of different I/O workloads under distinct CPU caps due to static and dynamic priorities. The results shown were the average of ten runs. A cap of 100 refers to a full pCPU capacity. In general, for the I/O only case, I/O performance initially stayed unaffected as the cap dropped until the cap fell below I/O’s CPU demand. For example, I/O performance dropped at cap 60 for the netperf TCP test.

I/O Performance due to static priority. Figure 5.8 shows I/O performance while the I/O task was set to real-time priority. It suggests that I/O suffered significant performance loss when co-located with the CPU-bound task even with a strictly higher priority (denoted as I/O+CPU). I/O had as much as 27.4%, 57.2%, and 77% throughput loss, and 417x
Figure 5.10: \texttt{xBalloon} improves the performance of Web server and database applications under constrained CPU allocation.

latency hike for netperf TCP, UDP, disk I/O, and sockperf, respectively. As scheduling was not based on the vruntimes of the two tasks under different priorities, no short-term priority inversion happened and the performance loss was due to long term priority inversions. Therefore, \texttt{rq\_clock\_task} did not help in I/O performance. In contrast, \texttt{xBalloon} achieved near-native performance for all four I/O workloads compared to case I/O only. As CPU cap approached to the I/O demand or fell below it, e.g., TCP throughput under cap 60 in Figure 5.8(a) or disk throughput under cap 20 in Figure 5.8(c), \texttt{xBalloon} incurred performance drop compared to the reference performance. This is due to the balloon’s CPU consumption, which together with I/O’s CPU demand, goes beyond the CPU cap. Further, our results also show that \texttt{xBalloon} significantly reduces the variation of sockperf latency compared to I/O+CPU.

I/O Performance due to dynamic priority. Figure 5.9 shows the results on fair sharing enforced by CFS dynamic priorities. All tasks including the balloon were assigned the same policy \texttt{SCHED\_OTHER} and scheduled by Linux CFS. Compared to Figure 5.8, I/O performance with \texttt{xBalloon} was worse than the reference I/O performance due to fair sharing CPU resources among tasks, e.g. UDP throughput under cap 40 in Figure 5.9(b). Another observation in Figure 5.9 is that \texttt{rq\_clock\_task} helped in I/O performance in some cases. For example, for netperf TCP in Figure 5.9(a), it improved throughput over
Figure 5.11: xBALLOON starves the compute task when CPU cap is lower than the I/O demand and allows the compute task to use slack CPU resources when CPU cap increases.

I/O+CPU by as much as 12%. The improvement was due to the avoidance of short-term priority inversions. However, rq_clock_task was not effective for performance loss due to long-term priority inversions. In contrast, xBALLOON improved I/O performance over rq_clock_task on average by 48.5%, 57.4%, 95.3%, and 125x for netperf TCP, UDP, disk I/O, and sockperf latency, respectively.

Note that xBALLOON even outperformed I/O only in some tests, e.g., TCP throughput under cap 40 in Figure 5.9(a). Similar results can also be observed in CPU sharing and the reason will be revealed in § 5.6.2.1.

5.6.1.2 Application Results

Next, we evaluate xBALLOON using real world I/O-intensive applications. If not otherwise stated, the compute task was mcf from the SPECCPU 2006 benchmarks [29] and the VM had a cap of 50.

**Web server applications.** We hosted a Nginx [23] HTTP server in the VM and ran Siege [27] on another machine as the client. We simulated 200 users that sent a total of 4,000 requests. As shown in Figure 5.10(a) and (b), Nginx suffered significant performance loss when co-running with the compute-bound program. In contrast, xBALLOON improved the throughput and 99th percentile latency by 14.4%, 14x under static priority, and by 33.3%, 10x under dynamic priority, respectively.

**Database applications.** First, we evaluated the performance of Sysbench [30], an OLTP application benchmark running on a MySQL database. We created 10 MySQL tables and each table contained 100,000 records. A client which contained 100 threads performed OLTP transactions in the database. As shown in Figure 5.10(c), xBALLOON increased Sysbench transaction rate by 32.9% under static priority and 20% under dynamic priority in
comparison with $I/O+CPU$, respectively. Second, we tested NoSQL database applications. We ran Redis [26] 3.0.7 as a server and used redis-benchmark to send 100,000 requests from a client. As shown in Figure 5.10(d), xBALLOON achieved close performance to $I/O$ only and outperformed $I/O+CPU$ by 90.2% under static priority and 136% under dynamic priority, respectively.

### 5.6.1.3 Preserving Static and Dynamic Priorities

It is challenging to directly verify priority preservation. The compute task’s CPU usage reported by Linux may be inaccurate when using the global clock $rq\_clock$. It can contain the time the VM is paused by xBALLOON. Instead, we used two indirect approaches.

**Enforcing static priority.** We changed the CPU hog to loop for a specified number of iterations and used its completion time to infer its CPU allocation. Figure 5.11 shows its completion time under various CPU caps. The completion time in $I/O+CPU$ is the baseline, in which the CPU hog acquired excessive CPU due to work-conserving scheduling in the guest. When fair sharing was enabled (white bar), the completion time of the compute task increased, indicating that the compute task was allocated less CPU. When static priority was enforced (red bar), the compute tasks failed to complete under low CPU caps (i.e., the missing data pointed by the arrow in Figure 5.11). xBALLOON enforced strictly higher priority for the I/O task and starved the CPU task when the cap cannot fully satisfy I/O’s demand. As cap increased, the CPU task was able to use slack CPU and complete, but with longer completion time compared to that in fair sharing.

**Preserving dynamic priority.** Although Figure 5.11 qualitatively demonstrates the preservation of static priority, it cannot quantitatively verify that the two tasks were indeed fairly scheduled under dynamic priority. I/O processing in the guest kernel including that in the
soft and hard interrupt contexts is charged to the CPU task if it happens to be the one running when I/O requests arrive. To this end, we emulated the I/O task’s CPU demand using a user-level process, which indefinitely repeats the cycle of computing and idling, whose ratio determines the CPU demand. We set the emulated I/O task to demand 30% of CPU. Since the process does not incur kernel processing, we use it to study how xBALLOON helps preserve dynamic priority.

Figure 5.12 shows the CPU allocations with (right bar) and without (left bar) xBALLOON under various cap settings. From the left bars in each group, we can see that the I/O task can only take 30% of the CPU time left by the cap no matter what was the new fair share based on the cap value. In contrast, with xBALLOON, CFS included future CPU allocations (i.e., the balloon) in fair sharing. CFS effectively satisfied the I/O demand in full when the cap was large and enforced the fair share between the two tasks as CPU cap dropped. We conclude that xBALLOON preserves dynamic priority by initially granting the I/O task a higher priority and smoothly demoting it to an equal priority.

Varying number of CPU workloads. In this experiment, we co-locate an I/O task with a varying number of CPU workloads. We measured TCP throughput and the 99th tail latency using sockperf. Figure 5.13(a)-(d) show I/O performance when the sockperf server process ran with one to four CPU hogs. Under static priority, as shown in Figure 5.13(a)
Figure 5.14: xBALLOON improves I/O performance with one I/O-bound VM and one CPU-bound VM sharing the CPU.

and (c), xBALLOON was always able to prioritize the I/O task. TCP throughput and latency were consistent despite the increasing levels of contention from co-located compute tasks. Under dynamic priority, the relative importance of the I/O task and the compute tasks are determined by their CPU usage. As shown in Figure 5.13(b) and (d), I/O throughput gradually dropped and latency increased as the number of CPU hogs increased. When there were only two tasks sharing the CPU, i.e., the I/O task and one compute task, the CPU demand of the sockper server process is lower than the fair share, i.e., 50% of CPU. Thus, xBALLOON prioritized the I/O task and achieved similar performance compared to that under static priority. As CPU competition ramped up, the CPU allocation to each task decreased and the demand of the I/O task gradually exceeded the fair share. To enforce max-min fairness, xBALLOON assigned an equal priority to the I/O task. The degraded I/O performance was due to a decreasing fair share and CPU allocation to the I/O task. Nevertheless, xBALLOON substantially improved I/O performance under both static and dynamic priorities compared to the I/O+CPU case.
Figure 5.15: xBALLOON improves I/O performance with two I/O-bound VMs sharing the CPU.

5.6.2 CPU Sharing

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

5.6.2.1 Sharing with CPU-bound VM

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

Figure 5.14(a) and (b) show the netperf TCP/UDP throughput under static and dynamic priorities. xBALLOON improved TCP and UDP throughput over I/O+CPU. The improvement on TCP throughput was 18.6% and 34.4% under static and dynamic priority, respectively. Similar to our observations in Figure 5.9(a) under CPU capping, xBALLOON with dynamic priority outperformed the reference I/O performance in I/O only. The reason is that when the I/O task ran alone or ran in a real-time priority with xBALLOON, it used as much time as it can until the entire VM is suspended due to CPU cap or descheduled due to insufficient credits. The VM was suspended for one accounting period (30ms in Xen) or descheduled for one time slice (also 30ms in Xen) before next credit refill.

In contrast, when the I/O task ran with xBALLOON under CFS fair sharing, it did not fully exercise its CPU demand. The VM can maintain a utilization no larger than the CPU cap or never used up its credit so as to enter the OVER state, either of which avoided long time freeze of the VM. This finding suggests that temporarily throttling I/O demand under constrained CPU allocation can lead to superior performance. As shown in Figure 5.14(c)
<table>
<thead>
<tr>
<th>Experiment</th>
<th>rps</th>
<th>avg_lat</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>std</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O only</td>
<td>5010.3</td>
<td>0.167</td>
<td>0.192</td>
<td>0.204</td>
<td>0.251</td>
<td>0.094</td>
</tr>
<tr>
<td>I/O+CPU(S)</td>
<td>5015.6</td>
<td>38.053</td>
<td>79.311</td>
<td>80.393</td>
<td>81.152</td>
<td>31.418</td>
</tr>
<tr>
<td>I/O+CPU+xBALLOON(S)</td>
<td>4997.7</td>
<td>0.169</td>
<td>0.197</td>
<td>0.211</td>
<td>0.263</td>
<td>0.097</td>
</tr>
<tr>
<td>I/O+CPU(D)</td>
<td>5023.1</td>
<td>39.352</td>
<td>80.128</td>
<td>82.336</td>
<td>83.713</td>
<td>32.926</td>
</tr>
<tr>
<td>I/O+CPU+xBALLOON(D)</td>
<td>4998.3</td>
<td>0.164</td>
<td>0.190</td>
<td>0.204</td>
<td>0.259</td>
<td>0.114</td>
</tr>
</tbody>
</table>

Table 5.1: The throughput (request per second) and latency (ms) of the caching benchmark in various test scenarios. \( S \) and \( D \) denote static and dynamic priority for the I/O workload, respectively.

and (d), \( x\text{BALLOON} \) achieved almost identical latency distribution compared to the reference latency distribution while \( I/O+CPU \) had wildly growing 75th percentile latency.

## 5.6.2.2 Sharing between I/O-bound VMs

The most challenging scenario is to consolidate two VMs, each running a mix of I/O- and CPU-bound workloads, and to preserve I/O prioritization in each VM. As the two VMs shared the Gigabit NIC on the host, the network would be the bottleneck if throughput tests were performed. Thus, we measured the tail latency on each VM using \texttt{sockperf} UDP test. The CPU-bound tasks were \texttt{mcf}. Figure 5.15(a) and (b) show the latency distribution of the two VMs. With \( x\text{BALLOON} \), both VMs had predictable and low tail latency close to the reference \( I/O \ only \) case. The latencies were not affected by co-running compute tasks, showing a clear preservation of I/O prioritization in both VMs.

## 5.6.3 Results on SMP VMs

This section presents an evaluation of \( x\text{BALLOON} \) with SMP VMs. We are interested in evaluating the effectiveness of \( x\text{BALLOON} \) in preserving I/O prioritization for multi-threaded workloads and studying its impact to the fairness of SMP CPU allocation and the overall system utilization. We configured two 4-vCPU VMs to share a set of 4 pCPUs and pinned each vCPU in a VM to a separate pCPU. As a result, two vCPUs from different VMs compete for the same pCPU in a time-sharing manner. We used the multi-threaded Data Caching benchmark from Cloudsuite [5] as the latency-sensitive, I/O-bound workload and \texttt{mcf} as the compute-bound workload. The Data Caching benchmark uses \texttt{memcached} [21] configured with 4 threads to simulate the behavior of a Twitter caching server using the twitter dataset. Two client machines, each with 4 worker threads and 20 connections were used to send requests at a rate of 5000 per second. The ratio of \texttt{GET} and \texttt{SET} was set to 4:1. To fully utilize four vCPUs, four copies of \texttt{mcf} were run in each VM.
Figure 5.16: xBALLOON does not cause significant idleness in the host and preserves fairness in CPU allocation.

Since the caching benchmark is network intensive, the 1Gbps Ethernet on the host machine was the bottleneck when two VMs both ran memcached. Thus, we measured the performance of multi-threaded I/O in one VM while the other VM only executed the compute-bound programs. Table 5.1 shows the throughput (request per second) and latency (ms) of memcached. I/O only was the case when one VM was dedicated to memcached and the other VM hosted the compute-bound workload. Because Xen prioritizes I/O workloads at the hypervisor level, the consolidation of I/O- and CPU-bound workloads in separate VMs does not affect I/O performance, we regard I/O only as the reference I/O performance. Similar to previous observations, the co-location of memcached and mcf (denoted as I/O+CPU), in addition to sharing pCPUs with another compute-bound VM, inflicted significant I/O performance degradation. While throughput was not much affected, the average and tail latency were drastically increased by several orders of multitude under both static and dynamic priority. To preserve I/O prioritization, we enabled four balloon processes, each bound to a vCPU, in the memcached VM. Results (in bold font) show that xBALLOON effectively preserved both static and dynamic priorities and achieved performance close to the reference performance.

Figure 5.16 shows the CPU utilisations of the two VMs when they were both running a mixture of memcached and mcf with xBALLOON enabled in both VMs. Recall that xBALLOON switches to NWC mode at the beginning of each accounting period. Thus, it is possible that both VMs could schedule the balloon process and yield CPU at the same time, causing idleness in the host machine. Figure 5.16 show that it took approximately 6s for the CPU allocation to converge to the fair share (i.e., 200%, half of the 4-pCPU capacity) and no significant CPU idling was observed in the host. As discussed in § 5.4.3, it is unlikely that both VMs yield CPU. Even if this happens, the host only experiences
transient idleness and xBALLOON helps to coordinate the scheduling of the two VMs such that their busy and idle (due to the balloon) periods are interleaved.

5.6.4 Discussion

**Overhead.** xBALLOON’s overhead includes the time spent in user space, in the guest kernel, and inside Xen. As the balloon is woken up by I/O events, the frequency of xBALLOON invocation depends on the intensity of the I/O workload. The overhead is negligible in most cases. As CPU cap or share drops close to the actual I/O demand, xBALLOON can lower the CPU available to serving I/O, leading to moderate degradation of I/O performance.

**Target I/O workloads.** For most I/O-bound applications, xBALLOON can effectively improve its throughput or latency when co-locating with CPU-intensive applications under discontinuous time. However, xBALLOON is not effective for workloads with both substantial I/O and computational requirements. The reason is that the priority inversion issue is not significant between compute-intensive I/O workloads and real compute-bound workloads, thereby leaving little room for improvement.

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

5.6.5 Results on Amazon EC2

Last, we show the effectiveness of xBALLOON on Amazon EC2. Even though we do not have access to the hypervisor, xBALLOON can still pause an EC2 instance. We used the hypercall SCHEDOP_block to pause the VM and disabled SWC scheduling. We built
a Amazon Machine Image (AMI) with our modified Linux kernel on an m3.medium instance, whose network bandwidth is capped at 340 Mbps and CPU at about 60%. Thus, we can only test on network latency. Figure 5.17 shows the results on TCP 99th percentile latency using sockperf and the average latency of ping. Since Linux network stack directly responds to ping’s ICMP requests, it is not possible to assign real-time priority to a user space process. Thus, we skipped real-time results for ping. The figure shows that xBALLOON effectively reduced TCP tail latency by 68% and even achieved a lower latency in ping test than the I/O only case. Although the test lacks important optimizations for xBALLOON, it shows that wiser scheduling inside the guest, without undermining the autonomy of the VM or changing resource management at the hypervisor, can greatly improve I/O performance.

5.7 Conclusion

This work demonstrates that task scheduling in the guest OS should be adapted to efficiently utilize virtual CPU resources. Time discontinuity due to CPU multiplexing or capping can render I/O prioritization in the guest ineffective, leading to I/O performance loss and unpredictability. This work presents xBALLOON, a lightweight approach to preserving static and dynamic priorities between I/O- and compute-bound tasks. xBALLOON centers on two designs: a CPU balloon representing CPU reservations and semi-work-conserving scheduling in VM. We demonstrate that xBALLOON is effective in boosting I/O performance and preserving priorities in both CPU capping and sharing.
CHAPTER 6
CHARACTERIZING AND OPTIMIZING HOTSPOT PARALLEL GARBAGE COLLECTION ON MULTICORE SYSTEMS

The proliferation of applications, frameworks, and services built on Java have led to an ecosystem critically dependent on the underlying runtime system, the Java virtual machine (JVM). However, many applications running on the JVM, e.g., big data analytics, suffer from long garbage collection (GC) time. The long pause time due to GC not only degrades application throughput and causes long latency, but also hurts overall system efficiency and scalability.

In this work, we present an in-depth performance analysis of GC in the widely-adopted HotSpot JVM. Our analysis uncovers a previously unknown performance issue – the design of dynamic GC task assignment, the unfairness of mutex lock acquisition in HotSpot, and the imperfect operating system (OS) load balancing together cause loss of concurrency in Parallel Scavenge, a state-of-the-art and the default garbage collector in HotSpot. To this end, we propose a number of solutions to these issues, including enforcing GC thread affinity to aid multicore load balancing and designing a more efficient work stealing algorithm. Performance evaluation demonstrates that these proposed approaches lead to the improvement of the overall completion time, GC time and application tail latency by as much as 49.6%, 87.1%, 43%, respectively.

6.1 Introduction

Due to its ease of use, cross-platform portability, and wide-spread community support, Java is becoming popular for building large-scale systems. Many distributed systems, such as Cassandra [50], Hadoop [13], Kafka [87], and Spark [139], are written in Java. Furthermore, there is also a steady trend towards adopting similar managed programming languages in high performance computing (HPC) [114, 147, 166]. Garbage collection (GC) is a crucial component of the automatic memory management in managed runtime systems, e.g., the Java Virtual Machine (JVM). It frees up unreferenced memory in the heap such that programmers do not need to concern about explicit memory deallocation. However, many studies [68, 69, 116] have shown that the widely adopted, throughput-oriented GC design suffers from suboptimal performance and poor scalability on the multicore systems with large memory and high core count.
Throughput-oriented GC pauses mutators, i.e., application threads, during GC to avoid expensive synchronizations between the GC and mutator threads. This period is called a stop-the-world (STW) pause. Since mutators cannot make progress during a STW pause, GC time can contribute to a non-trivial portion of application execution time. Previous work has shown that GC can take up to one-third of the total execution time of an application [67, 68]. It can even account for half of the processing time in memory-intensive big data systems [70, 116]. The exceedingly long GC time hurts system throughput and incurs unpredictable and severely degraded tail latency in interactive services [62, 103].

Parallel GC employs multiple GC threads to scan the heap and is designed to exploit hardware-level parallelism to reduce STW pause time. However, many studies have reported inefficiency and poor scalability of parallel GC on multicore systems. Existing studies [68, 69, 78, 156, 124] focus on optimizing the parallel GC algorithm in the JVM and assume that the underlying operating system (OS) provides the needed parallelism to execute parallel GC. There has been much research on analyzing the scalability of multi-threaded applications based on this assumption. We found that OS thread scheduling, particularly multicore load balancing, can have substantial impact on parallel GC performance. Our experiments with OpenJDK 1.8.0 and the Parallel Scavenge (PS) garbage collector revealed that many representative Java applications, including programs from the DaCapo, SPECjvm2008, and HiBench big data benchmarks, are unable to fully exploit multicore parallelism during GC. The main culprit is the uncoordinated design of the JVM and the underlying multiprocessor OS. On the one hand, modern OSes have complex load balancing algorithms due to the consideration of scalability, data locality, and energy consumption. Depending on different types of workloads, the OS thread scheduler needs to strike a balance between grouping threads on a few cores and distributing them on many cores. On the other hand, the JVM, which assumes perfect OS load balancing, has its own design for efficient load balancing among GC threads and synchronization primitives used within the JVM, e.g., mutex.

In this work, we identify two vulnerabilities in the HotSpot JVM and Parallel Scavenge due to the lack of coordination with OS-level load balancing. First, Parallel Scavenge implements dynamic GC task assignment to balance load among GC threads, but uses an unfair mutex lock to protect the global GC task queue. Although the unfairness is necessary for minimizing locking latency and believed to be harmless to GC performance, it inadvertently limits the concurrency in parallel GC when the underlying OS load balancing is “imperfect” and some GC threads are stacked on the same core. In this case, one or a few GC threads, which are able to continuously fetch GC tasks, will block other GC threads, leaving much of multicore parallelism unexploited. Since the unfair mutex implementation is also used for synchronizing VM threads and mutators, this problem may also exist in many user space Java applications. Second, to further balance GC load, an
idle GC thread steals work from a randomly selected GC thread. A steal attempt fails if the selected GC thread has no extra work to be stolen. This lack of coordination between the JVM and the multicore OS causes the heuristics that guide work stealing to be ineffective, which delays the termination of the GC.

To address these vulnerabilities, we propose two optimizations in the HotSpot VM. Through an in-depth analysis of the effect of unfair locking on GC performance and the evaluation of two fixes to the unfairness issue in the JVM mutex, we find that GC thread affinity, which dynamically binds GC threads to separate cores based on CPU load, is effective in preventing load imbalance among GC threads. To address the inefficiency in GC work stealing, we devise an adaptive stealing policy that dynamically adjusts the number of steal attempts according to the number of active GC threads and improves steal success rate using a semi-random stealing algorithm. Our experiments with industry-standard benchmarks, DaCapo and SPECjvm2008, and two real-world Java applications, Cassandra database and HiBench big data benchmarks, show up to 49.6%, 87.1% and 43% improvement on application execution time, GC time and request tail latency, respectively, due to our optimizations on parallel GC. To summarize, this work makes the following contributions:

- **In-depth analysis of GC performance in multicore systems.** We leverage comprehensive GC profiling, knowledge of OS scheduling and thread synchronization to identify vulnerabilities in the Parallel Scavenge and the HotSpot JVM that can inflict a loss of concurrency during parallel GC. The resulted load imbalance significantly prolongs the STW pause time.

- **Proposing two optimizations to address GC load imbalance.** We discuss our attempts to addressing GC load imbalance and propose a dynamic GC load balancing scheme with coordination between the JVM and the OS kernel. We further improve GC load balancing by designing an adaptive and semi-random work stealing algorithm inside the JVM.

- **Comprehensive evaluations of the proposed optimizations.** Our evaluations on DaCapo, SPECjvm2008, Cassandra, and HiBench show considerable and consistent improvement on parallel GC. We also demonstrate that our optimizations improve the performance on application scalability, various heap configurations as well as in complex application execution environments,

6.2 Background

6.2.1 Parallel Scavenge

Garbage Collection is the process of automatically freeing objects that are no longer referenced by application threads (mutators). It scans root references in the heap and records
references that are reachable during the scan. Objects with unreachable references are regarded as garbage and are reclaimed in a sweep phase. Parallel Scavenge uses a stop-the-world design, which pauses mutator threads until GC completes. The collection involves three phases: initialization phase, parallel phase, and final synchronization phase [68].

In the initialization phase, the VM thread ensures that all mutator threads are suspended before waking up GC threads. After the GC threads become live, the VM threads sleep and wait for the final phase. Collection is performed in the parallel phase, in which the GCTaskManager creates and adds GC tasks into the GCTaskQueue from where multiple GC threads can fetch and execute them in parallel. With the help of the global task queue, Parallel Scavenge implements dynamic task assignment among GC threads (Section 6.2.2).

Parallel Scavenge performs generational garbage collection [44] by dividing the heap into multiple generations: young, old, and permanent generation. The young generation is further divided into one eden space and two survivor spaces, i.e., from-space and to-space. When the eden space is filled up, a minor GC is performed. Referenced objects in eden and from survivor space are moved to the to survivor space, and unreferenced objects are discarded. After a minor GC, the eden and the from space are cleared, and objects survived in the to space have their age incremented. After surviving a predefined number of minor GCs, objects are promoted to the old generation. Similarly, as the old generation is filled up, a major GC is triggered to free space in the old generation. Both minor and major GCs obtain tasks from GCTaskQueue except that GCTaskManager prepares different GC tasks for them. Among GC tasks, steal tasks are used to balance load between GC threads and are always placed after normal GC tasks in GCTaskQueue. GC threads that have fetched steal tasks attempt to steal work from other GC threads. When all GC threads complete the parallel phase and suspend themselves, the VM thread is woken up, entering the final synchronization phase. After resizing the generations based on the feedback of recently completed GCs, the VM thread wakes up the mutators and suspends itself until the next GC.

6.2.2 Dynamic GC Task Assignment

Figure 6.1 shows the implementation of dynamic GC task assignment during the parallel phase of Parallel Scavenge. At the beginning of the parallel phase, GCTaskManager adds various types of GC tasks, e.g., OldToYoungRootTask, ScavengeRootsTask, ThreadRootsTask, and StealTask, to the GCTaskQueue. As these tasks may contain different amounts of work, the load assigned to GC threads can be unbalanced. Dynamic task assignment, which only sends a task to a GC thread when it requests one, helps resolve the imbalance as GC threads assigned with smaller tasks would fetch more. To prevent concurrent access to GCTaskQueue, GCTaskManager is implemented as
a monitor, which can only be owned by one GC thread at a time. GC threads keep attempting to fetch (i.e., get_task) and execute a task each time. Multiple GC threads are synchronized by a monitor-based GC task manager. If the queue is empty, i.e., all GC tasks have been completed, a GC thread suspends itself to the WaitSet in the monitor. Threads sleeping in the WaitSet can later be woken up when new tasks are added to the task queue, i.e., when the next GC begins. The waking GC threads compete for the mutex lock before they can dequeue a GC task.

### 6.2.3 Work Stealing among GC Threads

To further balance load, a GC thread can steal work from another thread if it would otherwise stay idle. Once a GC task is fetched from the task queue, a GC thread divides it into many fine-grained tasks and pushes them into a local task queue, i.e., GenericTaskQueue in Figure 6.1. Such finer-grained tasks can be stolen by others. For example, a root task in the young-generation collection pushes every reference it accesses in the object graph to the local breadth-first-traversal queue, in which each reference leading to a sub-graph is a fine-grained task.

Parallel Scavenge places steal tasks, one for each GC thread, after ordinary GC tasks in GCTaskQueue. Therefore, if no ordinary tasks are available in the queue, GC threads fetch steal tasks and start work stealing. A GC thread enters the final synchronization phase when GCTaskQueue is empty and there is no task to be stolen from other GC threads. Parallel Scavenge uses a distributed termination protocol to synchronize GC threads. After $2 \times N$ consecutive unsuccessful steal attempts, a GC thread enters the termination procedure, where $N$ is the number of GC threads. It atomically increments a global counter.

---

1HotSpot uses a heuristic to determine the number of GC threads: $N = (ncpus \leq 8) ? ncpus : 3 + ((ncpus * 5)/8)$, here ncpus denotes the number of CPU cores.
Figure 6.2: Native monitor in the HotSpot JVM.

_offered_termination to indicate termination. If the counter reaches $N$, all GC threads have terminated and the parallel phase ends. While in the termination protocol, a GC thread periodically peeks if there are any root tasks available from any of the GC threads. If so, it decrements the counter and returns back to stealing. The core of a steal attempt is the function steal_best_of_2 that selects two randomly chosen GC threads and steals tasks from the one with the longer queue [113].

6.2.4 The Implementation of Monitor in HotSpot

Monitor is a synchronization mechanism that contains a condition variable and its associated mutex lock. It allows threads to have mutual exclusive access to a shared data structure and to wait on a certain condition. Figure 6.2 shows the structure of the native monitor in HotSpot. Parallel Scavenge implements GCTaskManager as a monitor to protect GCTaskQueue. When GCTaskQueue is empty, either before the first GC or at the end of the previous GC, all GC threads sleep in WaitSet. GCTaskManager notifies and wakes up all GC threads when the next GC begins.

The critical design in monitor is the mutex lock, which should strike a balance between efficiency and scalability. To this end, HotSpot implements two paths, fast and slow, for lock acquisition. A thread acquires the ownership of a mutex by changing the LockByte in the mutex from zero to non-zero using an atomic compare-and-swap (CAS) instruction. On the fast path, a thread first attempts to CAS the LockByte. If the mutex is not contended, lock acquisition is successful. Otherwise, the thread turns to the slow path. Although a CAS fast path offers low locking latency for a small number of threads, it incurs considerable cache coherence traffic on a many-core system with a large number of threads [100].

The slow path is designed for scalability. It contains two separate queues for lock contenders: cxq and EntryList and two internal lock states: OnDeck and owner.
Recently-arrived threads push themselves onto cxq if the fast path fails. In addition, GCTaskManager, at the beginning of each GC, transfers sleeping GC threads from the WaitSet of the monitor to cxq, letting waking GC threads compete for the mutex. Owner is the current lock holder and OnDeck is the thread selected by the owner as the presumptive heir to acquire the lock. The OnDeck thread is promoted from the EntryList. If EntryList is empty, owner moves all threads on cxq to EntryList. Both queue promotion and heir selection are performed by the lock owner when it unlocks the mutex.

This slow path is efficient for highly contended mutex. It throttles concurrent attempts to acquire the lock from a large number of threads. For example, GC threads are transferred from WaitSet to cxq without being woken up to avoid severe contention from multiple CAS attempts. Furthermore, threads on cxq or EntryList are in the sleep state and not allowed to attempt lock acquisition. HotSpot ensures that there can be at most one OnDeck thread. Thus, at any time, there are at most three (types of) contenders on the lock: the OnDeck thread, the owner that just released the lock, newly arrived thread(s) which has not been placed on cxq. To avoid the lock-waiter preemption problem [120], in which a thread supposed to acquire the lock is preempted, delaying other waiters, HotSpot uses a competitive handoff policy. Instead of directly passing the lock from the owner to the OnDeck thread, the owner wakes up the OnDeck thread and lets it compete for the lock by itself. As such, even if OnDeck is preempted, other threads are still able to
acquire the lock through the fast path. Although the above mutex design provides excellent throughput, it sacrifices short-term fairness: 1) it allows the owner thread to re-acquire the mutex lock, possibly causing starvation to the lock waiters on cxq and the OnDeck thread; 2) newly-arrived threads can bypass the queued lock waiters. We will show in § 6.3 that the short-term unfairness can cause severe inefficiency in the parallel phase of Parallel Scavenge.

### 6.2.5 Linux Load Balancing

Load balancing is a critical component in an OS scheduler. By evenly distributing threads on all cores, it minimizes the average queuing delay on individual cores and exploits the parallelism on multicore hardware. However, load balancing has become very complex in modern OSes due to the consideration of overhead, scalability, data locality, and power consumption. In this section, we describe the load balancing algorithm in Linux’s Completely Fair Scheduler (CFS), the widely used OS scheduler in production systems.

Due to scalability concerns, CFS uses per-core run queues on a multicore machine. Individual cores are responsible for time sharing the CPU among multiple threads. Load balancing is implemented by means of thread migration across cores. Overall, CFS tracks load on each core and transfers threads from the most loaded core to the least loaded. Since thread migrations between cores require inter-core synchronization, load balancing should not be performed too frequently to avoid high overhead. In general, there are three scenarios that trigger load balancing: 1) a core becoming idle for the first time will attempt to steal runnable threads from the run queue of a busy core; 2) a core periodically runs the balancing algorithm; 3) a waking thread can be migrated from its current core to the idlest core in the system.

Load balancing can be ineffective for several reasons. First, Linux only migrates “runnable” threads between cores. Thus, GC threads that frequently sleep may miss either idle balancing (scenario 1) or periodic load balancing (scenario 2). In addition, the load balance interval in CFS is coarse grained compared to the length of GC tasks. For example, the default interval for periodic load balancing between two hyperthreads is $64ms$ and the interval increases (multiply by 2) as the distance between the CPUs increases. In comparison, for most applications, the GC should complete within a few hundreds of milliseconds to avoid a long pause of the application threads. Therefore, each individual GC task typically lasts a few tens or hundreds of microseconds. Third, CFS avoids waking up idle cores that are in a deep sleep state for load balancing to save energy. In any of these circumstances, multiple GC threads can be stacked on a few cores even when there are idle cores in the system. Therefore, load balancing in CFS is most effective with workloads with a stable degree of parallelism but fails to function properly with GC threads that exhibit dynamic parallelism.
6.3 Analysis of Inefficient Parallel GC

This section presents an in-depth analysis of the performance of Parallel Scavenge on a multicore machine. We first show the poor scalability of parallel GC and its impact on application performance. Then, we attribute the suboptimal GC performance to load imbalance among GC threads and inefficient stealing.

6.3.1 Parallel GC Performance and Scalability

We selected four workloads, two scalable workloads *xalan* and *lusearch* from the DaCapo benchmarks [58], *kmeans*, a well-known clustering algorithm for data mining from the HiBench big data benchmarks [80], and *Cassandra*, a distributed NoSQL database. For the traditional JVM workloads, such as *xalan* and *lusearch*, the JVM heap size was set to three times of the minimum heap requirement [79]. *Kmeans* and *Cassandra* had a heap size of 4GB and 8GB, respectively. All benchmarks were executed on Linux 4.9.5, OpenJDK 1.8.0 and Parallel Scavenge. The experiments were tested on a Dell PowerEdge T430 with dual Intel 10-core processors. Details of the testbed and benchmark settings can be found in Section 6.5.1.

Figure 6.3 (a) shows the performance of *xalan* and *lusearch* with various numbers of mutator threads and a breakdown of mutator execution time and GC time. Parallel Scavenge sets the number of GC threads to 15 on our 20-core machine. As shown in Figure 6.3 (a), in *xalan* and *lusearch*, mutator time dropped as the number of mutators increased and thus GC time became more significant in the overall execution time. For instance, GC contributed to 43.2% of the total time in the case of 16 mutator threads, incurring unacceptable overhead to application performance. Next, we evaluated the performance of *kmeans* in Spark. Figure 6.3 (b) shows the performance with a varying number of mutators and two input sizes: small and large. Similar to the DaCapo results, the ratio of GC time increased as mutator time decreased. In addition, the large dataset incurred much higher GC overhead compared to the small dataset.

In Figure 6.3 (c), we fixed mutator threads to 16 and varied the number of GC threads to study GC scalability. For both *xalan* and *lusearch*, parallel GC scaled poorly with increasing parallelism. The GC time even ramped up as the number of GC threads increased. Another observation is that mutators had prolonged execution time with more GC threads. It suggests that inefficient GC not only hurts JVM memory management but also influences mutator performance. Figure 6.3 (d) studies request latency in the *Cassandra* database. We used another client machine executing a varying number of threads to read one million records from the database. The figure shows that the request latency increased exponentially with more intensive client traffic. The ratio of GC time in the total execution time also climbed to 25%. As an STW collector, the increased parallel GC time can significantly
prolong tail latency. In what follows, we identify the causes of the inefficient GC and its poor scalability.

### 6.3.2 Load Imbalance

We instrumented the HotSpot JVM to report two types of load information during parallel GC: 1) GC task distribution among GC threads and 2) GC thread distribution on CPU cores. We analyzed *lusearch* as it shows significant GC overhead and poor scalability in Figure 6.3. The number of mutator threads was set to 16 and the number of GC threads was automatically set by Parallel Scavenge to 15.

**Task Imbalance.** To monitor task distribution, we modified the constructor of GC tasks to log the GC thread ID on which a GC task is executed. According to the GC logs, parallel GC in *lusearch* is dominated by minor GC, which includes fifteen *OldToYoungRootsTasks*, nine *ScavengeRootsTasks*, thirty-four *ThreadRootsTasks* and fifteen *StealTasks*. Note that the number of *StealTasks* matches the number of GC threads. As shown in Figure 6.4 (a), GC tasks, except *StealTasks*, were unevenly distributed among GC threads. *Lusearch* incurred around 198 minor GCs during execution. The GC logs showed severe task imbalance in most minor GCs, though the ID(s) of overloaded GC thread(s) varied in each GC. The result clearly shows that Parallel Scavenge failed to exploit the available parallelism (i.e., 15 GC threads) in GC. Note that GC threads are homogeneous in Parallel Scavenge and do not have bias in task assignment, which led us to the investigation of GC thread execution in the underlying OS.

**Thread Imbalance.** To monitor GC thread execution, we traced function `GCTaskManager::get_task` and recorded the number of times each GC thread successfully dequeued a GC task from `GCTaskQueue` and on which CPU the GC thread was running. Figure 6.4...
Figure 6.5: The vulnerability of dynamic task assignment and unfair mutex lock in multicore systems.

(b) shows the execution of all GC threads of one minor GC in lusearch. It suggests that most GC threads were stacked on a few CPU cores while the remaining cores were idle. It is evident that multicore parallelism was not fully exploited. Figure 6.4 (b) also shows unbalanced GC task distribution with a few threads having fetched more tasks than others did.

**Root cause analysis.** As discussed in Section 6.2.5, Linux load balancing can be imperfect and temporarily place multiple GC threads on the same core. However, two critical questions remain unanswered: 1) **why is OS load balancing not effective during the entire GC?** 2) **why is time slicing/sharing on a single core not effective, otherwise stacking GC threads should have equal opportunities to fetch tasks and task imbalance should never occur?** We identified the reasons by monitoring the competitions between GC threads on the mutex lock that protects the GCTaskQueue. The GC log showed that throughout the GC, at any point in time, there were at most two GC threads (the OnDeck thread and the previous owner thread) actively competing for the mutex lock and the previous owner thread (almost) always won.

Figure 6.5 illustrates how the loss of concurrency in parallel GC develops. **First,** at the beginning of each GC, sleeping GC threads on the monitor's WaitSet are transferred to cxq and become waiters of the mutex lock. This is to prevent concurrent attempts on lock acquisition when all GC threads wake up at the same time. The monitor selects two threads at the head of cxq to be the lock owner and OnDeck, and the other threads remain blocked, not eligible for lock acquisition nor OS load balancing. Then, lock competition becomes a two-player game. **Second** and most importantly, the competition is unfair if the two threads are stacked on the same CPU. As shown in Figure 6.5, after the owner releases the lock and wakes up OnDeck, it executes the GC task it fetched from GCTaskQueue.
Once the task is completed, the previous owner thread executes `get_task` again, attempting to fetch another GC task and acquire the mutex lock. The fast lock acquisition path allows the previous owner to bypass waiters in cxq and EntryList and directly acquire the lock. If the two threads are on the same core, the OnDeck thread may never acquire the lock.

Most OS schedulers avoid frequent thread context switching on a CPU and guarantee that a thread can run for a minimum time before it is preempted. Therefore, after releasing the lock and waking up the OnDeck thread, the previous owner thread may continue to run on CPU if it has not used up its minimum time quantum. In this case, the waking OnDeck thread would fail to preempt the owner thread and be placed by the scheduler onto the CPU run queue as a runnable thread. Since the OnDeck misses the wakeup momentum to preempt the owner thread, it has to wait for a whole time slice before being scheduled. At the time the OnDeck thread is scheduled, if the owner thread has re-acquired the lock, the OnDeck thread would go to sleep again. This cycle repeats and the OnDeck thread may never acquire the lock until the owner thread depletes all GC tasks in the GCTaskQueue.

The stacking of GC threads will happen almost every time. When GC threads are first created by the JVM, they are spawned on one core. Since the GC task queue is empty at the launch time of the JVM, all GC threads will immediately block until the first GC begins. OS schedulers do not balance blocked threads, thus all GC threads are stacked on one core when the first GC starts, relying on OS load balancing to resolve the stacking. There are two practical obstacles to effectively balancing stacked GC threads.

First, during lock contention, there are at most two active GC threads, i.e., the owner thread and the OnDeck thread, that are eligible for load balancing. However, if the OnDeck cannot acquire the lock and remain in a blocked state, load balancing will not
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>Failure</th>
<th>Failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>237983</td>
<td>166008</td>
<td>69.8%</td>
</tr>
<tr>
<td>jython</td>
<td>75036</td>
<td>66318</td>
<td>88.4%</td>
</tr>
<tr>
<td>lusearch</td>
<td>117383</td>
<td>108308</td>
<td>92.3%</td>
</tr>
<tr>
<td>sunflow</td>
<td>45648</td>
<td>34695</td>
<td>76.0%</td>
</tr>
<tr>
<td>xalan</td>
<td>22783</td>
<td>19303</td>
<td>84.7%</td>
</tr>
<tr>
<td>compiler.compiler</td>
<td>726920</td>
<td>274043</td>
<td>37.7%</td>
</tr>
<tr>
<td>compress</td>
<td>29348</td>
<td>26513</td>
<td>90.3%</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>64493</td>
<td>60388</td>
<td>93.6%</td>
</tr>
<tr>
<td>xml.transform</td>
<td>457198</td>
<td>341555</td>
<td>74.7%</td>
</tr>
<tr>
<td>xml.validation</td>
<td>651475</td>
<td>188189</td>
<td>28.9%</td>
</tr>
</tbody>
</table>

Table 6.1: The total and failed steal attempts in steal_best_of_2().

take effect as there is no runnable thread to move. It is possible to decrease the minimum thread runtime to increase the chance of the OnDeck thread to preempt the owner thread, i.e., setting a smaller value for sched_min_granularity_ns in CFS. However, the tuning of CFS (i.e., setting the minimum thread runtime to 100µs) does not mitigate the unfairness in lock acquisition. At the beginning of each GC, the OnDeck thread is unlikely able to preempt the owner thread regardless of the minimum runtime as the owner also just woke up. After the first failed attempt, the OnDeck thread becomes runnable and has to wait a full time slice (12ms in CFS) to be scheduled. Once the owner thread is descheduled, its minimum runtime is reset. If the OnDeck thread fails to acquire the lock again, the vicious cycle of block, wakeup and failed lock acquisition continues.

**Second**, even if the OnDeck thread acquires the lock, the execution serialization still persists. The previously OnDeck thread becomes the owner thread and a sleeping lock waiter thread, which resides on the same CPU, will be promoted to OnDeck. Since the two new lock contenders are stacked on one CPU, the unfairness in lock acquisition still exists. Ideally, load balancing will be effective when all GC threads are active and visible to the load balancer. This requires that the non-critical section of GC, i.e., each GC task, be long enough to keep all GC thread busy at the time of load balancing. However, the amount of work in each GC task varies greatly, depending on the sparsity of the sub-graph reachable from a GC task. Some tasks can be quite small in a large heap. Therefore, it is almost impossible to keep all GC threads active all the time and thread stacking is inevitable. As a result, unfair locking in the JVM causes serialization among GC threads.

6.3.3 Ineffective Work Stealing

As discussed in Section 6.3.2, there exists significant load imbalance among GC threads. Next, we study the effectiveness of work stealing in addressing the imbalance.
Algorithm 1 Dynamic GC thread balancing

1: **Variable:** The load on the $i_{th}$ core $L_i$; the load degree of the $i_{th}$ core $D_i$; the average load across all cores $L_{avg}$; the number of cores with low load $N_{low}$; GC thread $t_i$.

2: /* Mark CPU load as: high, normal, and low. */

3: **function** Mark_CPU_Load_Degree

4: for each core $i$ do

5: if $L_i \geq 2 \times L_{avg}$ then

6: $D_i = \text{high}$;

7: else if $L_i \leq 0.5 \times L_{avg}$ then

8: $D_i = \text{low}$;

9: $N_{low}++$;

10: else

11: $D_i = \text{normal}$;

12: /* Dynamically rebalance GC threads to avoid contentions */

13: **function** GC_Thread_Rebalance($t_i$)

14: if $t_i$’s current CPU load degree is high then

15: $k = \text{rand}() \% N_{low}$;

16: bind $t_i$ to the $k_{th}$ core in the low load CPU set;

among GC threads. Figure 6.6 shows the breakdown of minor GC time of some representative applications in DaCapo and SPECjvm2008 [140]. We instrumented Parallel Scavenge to log the execution time of each GC stage. We further recorded detailed GC task completion time at each GC thread and divided the parallel GC phase into root task, steal task and steal termination. Note that the time breakdown in Figure 6.6 is aggregated among all GC threads. It is possible that when some threads were in steal termination, others were still executing root tasks. Thus, the GC time breakdown does not reflect the timeline of GC.

As shown in Figure 6.6, steal tasks dominate the total GC time in all benchmarks, which was also observed by Gidra et al. [67]. While a GC thread executes a steal task, it is either processing a task stolen from another thread or attempting a steal. In contrast, the time spent in the steal termination, during which terminated GC threads waiting for other active threads to synchronize at the barrier of the final phase, is wasted and does not contribute any meaningful computation. Table 6.1 lists the total and failed steal in steal_best_of_2. Most benchmarks suffered high failure rate except compiler.compiler and xml.validation. Recall that steal_best_of_2 selects the longer of two randomly chosen GC thread queues to steal. If the load is severely unbalanced, its performance can degrade to random stealing because most attempts return two empty queues. These observations motivated us to design a more efficient termination protocol and more effective stealing policy for steal tasks.
6.4 Optimizations

This section presents two optimizations of Parallel Scavenge to address its vulnerability and inefficiency in multicore systems (discussed in § 6.3). For each optimization, we describe its design and implementation, and evaluate its effectiveness on mitigating imbalance and wasteful stealing.

6.4.1 Addressing Load Imbalance

The culprits of load imbalance in parallel GC are ineffective OS load balancing of GC threads, unfair mutex acquisition, and dynamic task assignment that allows one or a few GC threads to deplete the GC task queue. To avoid re-designing the GC task model in Parallel Scavenge and changing dynamic task assignment, we explored optimizations to address unfair locking, such as disabling all fast paths in locking, enforcing fair (FIFO) mutex acquisition and allowing multiple active lock contenders. Unfortunately, without the help from the OS, these approaches either had no effect or led to degraded performance.

A simple approach to avoiding GC threads stacking is to disable the OS load balancing and pin GC threads to separate cores. BindGCTaskThreadsToCPUs has been included as a command line option in OpenJDK since version 1.4 but the backend function bind_to_processor was never implemented on Linux, Windows, or BSD. It was first proposed in specification JSR-282 but no agreement was made to provide the processor binding API due to lack of evidence for GC performance improvement and the difficulties to provide a generic API across various platforms [121]. The processor binding interface has been implemented in Solaris, but the benefits of binding was not well studied. To prove the necessity of GC thread affinity in parallel GC, we implemented this feature in OpenJDK 1.8.0 for Linux. When a GCTaskThread is created, it is bound to a core whose ID matches the thread ID.
Figure 6.8: Improved thread and task balance in *lusearch* minor GC with 16 mutators.

Static binding avoids stacking GC threads on cores but may conflict with other workloads scheduled by the OS scheduler. To address this issue, we devise a GC thread balancing scheme (Algorithm 1) to rebalance GC threads to avoid contentions with other workloads. At the start of each parallel GC, a GC thread examines the load on its current CPU and binds to a different CPU if the current one experiences contention. We leveraged `load_avg` in the Linux kernel to measure the load on each CPU. Note that `load_avg` only measures the load of ready/running tasks but does not count sleeping threads. This explains why OS load balancing is not effective for stacking GC threads as most of them are in the sleep state. To avoid stacking GC threads during the rebalancing, we incorporated the load from sleeping threads into `load_avg` in the Linux kernel.

As Algorithm 1 shows, a CPU is considered to have a *high* load if its load is higher than two times of the average load across all CPUs while a *low* load is less than half of the system-wide average (line 4-9). Each GC thread checks the load of its current CPU at the start of each GC and rebinds to a randomly picked CPU with low load (line 14-15). Figure 6.7 shows the steps to rebalance GC threads. The Hotspot JVM communicates with the Linux kernel via the `/proc` file system. The OS writes the run queue load of each CPU to the shared memory and the JVM reads the information when the `GCTaskManager` adds `GCTasks` to the `GCTaskQueue` and the GC threads are woken up (Step ①). The GC load analyzer determines whether it needs to rebind a GC thread to another CPU based on the load information (Step ②). If needed, our implemented `bind_to_processor` is used to rebind the GC thread (Step ③).

Besides the optimizations for GC thread balancing, we also made some modifications to improve the task-to-thread balance. The default `GCTask` has no task affinity even though the parameter `UseGCTaskAffinity` is enabled. We modified the constructor function of `OldToYoungRootsTask`, `ScavengeRootsTask` and `ThreadRootsTask` to include task affinity. When the `GCTaskThread` executes `get_task()`, `GCTaskManager` prefers to dequeue the task which has the affinity to that thread. If no task is found matching the affinity, `GCTaskManager` dequeues any task that is available.
Algorithm 2 Adaptive and semi-random work stealing

1: /* Adaptive termination protocol: only steal from active GC threads. */
2: Variable: The number of active GC threads $N_{live}$, the queue ID to steal $q$; the queue ID $q_s$ in last successful steal attempt.
3: function Steal_Task
4: for Steal attempts less than $2 \times N_{live}$ do
5: $q = \text{Steal\_Best\_of\_2}(q_s)$
6: if $q \neq \phi$ then
7: $q_s = q$
8: Steal from $q$ and return
9: $q_s = \phi$
10: Enter the termination protocol
11:
12: /* A semi-random algorithm for queue selection */
13: Variable: The queue ID $q_s$ in last successful steal attempt.
14: function Steal_Best_of_2($q_s$)
15: Randomly select the first queue $q_1$
16: if $q_s \neq \phi$ and $q_s$ is not empty then
17: $q_2 = q_s$
18: else
19: Randomly select the second queue $q_2$
20: if $q_1$ and $q_2$ are both empty then
21: $q_s = \phi$ and return $\phi$
22: else
23: return the longer of $q_1$ and $q_2$

We evaluated the effectiveness of the optimized thread and task balance design using lusearch from DaCapo, which was configured with 16 mutator threads and 15 GC threads. Figure 6.8 (a) shows that GC threads are evenly distributed on multiple cores. The warm temperature in the heatmap suggests that all GC threads were able to fetch tasks from GCTaskQueue. As shown in Figure 6.8 (b), GC thread and task affinity help mitigate load imbalance among GC threads. Compared to the case shown in Figure 6.4 (a), all GC threads are assigned with root tasks, showing much improved task balance among GC threads.

6.4.2 Addressing Inefficient Working Stealing

In Section 6.3.3, we identified two deficiencies of Parallel Scavenge’s work stealing algorithm: 1) the distributed termination protocol is slow and incurs too many steal attempts; 2) the queue selection algorithm is not effective, leading to high steal failure rate. In the original design, a GC thread enters the termination protocol after experiencing $2 \times N$ consecutive failed steal attempts, where $N$ is the number of GC threads. For each attempt, it
selects two random GC thread queues and steals from the longer one. We see two problems with such a design. First, the termination protocol requires $2N$ failures to end a GC thread regardless of how long the GC thread has been in the parallel GC. Towards the end of the parallel phase, most GC threads may have been in the termination protocol. Therefore, it is not necessary to wait for $2N$ failures to enter termination because there are only a few active threads, from which tasks can be stolen. Second, if load imbalance occurs, as we show in Figure 6.4 (a), work can be assigned to a few GC threads and the random stealing might be quite ineffective.

To address these two issues, we propose an adaptive and semi-random stealing algorithm, shown in Algorithm 2. We implemented a `FastParallelTaskTerminator` class in HotSpot to coordinate GC thread termination. It records the number of active GC threads (i.e., $N_{\text{live}}$) that are not yet in the termination protocol. As GC threads enter or exit the termination protocol, the active thread count is updated. A thread only steals from the pool of active GC threads (line 4-10). Accordingly, the criteria for thread termination becomes $2N_{\text{live}}$ consecutive failed steal attempts. The `steal_best_of_2` function was also modified to improve steal success rate. It memorizes the last queue from where the steal was a success and selects the same queue as one of the steal choices, given that the queue is not empty (line 19-20). Another queue is picked up randomly. Similarly, the longer of the two queues is chosen as the stealing target.

We evaluated the effectiveness of the optimized stealing algorithm using programs from DaCapo and SPECjvm2008. All programs ran with 16 mutator threads and 15 GC threads. As Figure 6.9 (a) shows, the optimized stealing reduced the total number of steal attempt for most of the benchmarks except `xml.validation`. Among these attempts, the portion of failed attempts, i.e., failure rate, also dropped, as shown in Figure 6.9 (b). While the reduction on steal attempts or failure rate alone is not significant, the aggregate benefit is clear. For example, the number of steals for `xml.validation` increased by 95.1% while the
failure rate dropped by 4.5x. As a result, the total number of failed attempts decreased by 56.8%, which indicates that the increased steal attempts contained mostly successful steals. Overall, the reduction on failed attempts ranged from 18.3% to 56.8%. As will be discussed in Section 6.5, the savings on futile steal attempts lead to improved GC performance.

### 6.5 Evaluation

In this section, we present an evaluation of our optimized JVM using various micro and application benchmarks. We first study the effectiveness of our design on improving the overall application performance (§ 6.5.2) and on reducing GC time (§ 6.5.3), and compare our work with GC optimization in other papers. We then analyze the impact of improved GC on application scalability (§ 6.5.4), and investigate how much our design improves the performance of real-world applications (§ 6.5.5) and applications with different heap configurations (§ 6.5.6). Finally, we demonstrate the performance improvement in a multi-application environment (§ 6.5.7) and discuss the effect of simultaneous multithreading(§ 6.5.8).
6.5.1 Experimental Settings

Hardware. Our experiments were performed on a DELL PowerEdge T430 server, which was equipped with dual ten-core Intel Xeon E5-2640 2.6GHz processors, 64GB memory, Gigabit Network and a 2TB 7200RPM hard drive. Initially, simultaneous multithreading (SMT) was disabled to isolate the effect of our proposed optimizations from interference on sibling hyperthreads. The heuristic used to determine the number of GC threads is based on CPU count. Enabling SMT on our testbed results in a total of 40 logical cores and 28 GC threads. Since the testbed only has 20 physical cores, 16 GC threads would run on sibling hyperthreads, which could either have constructive or destructive impact on Java programs [99, 82, 130, 108]. We enable SMT and study its effect in Section 6.5.8. The machine was configured with the default power management and all cores ran at their maximum frequency. Turboboost, mwait, and low power C-States were also enabled. For database benchmarks, we used another machine in the same Ethernet as the client.

Software. We used Ubuntu 16.10 and Linux kernel version 4.9.5 as the host OS. All experiments were conducted on OpenJDK 1.8.0 with Parallel Scavenge as the garbage collector. If not otherwise stated, we set the number of mutators to 16 and the decision on the number of GC threads was left to Parallel Scavenge, which created 15 GC threads for our 20-core testbed. This setting ensured that the machine is under-provisioned and both the mutators and GC threads had access to sufficient multicore parallelism.

Benchmarks. We selected a subset of programs in the following benchmarks and measured their performance on the vanilla JVM and the one with our optimizations. The selected workloads are scalable workloads that benefit from parallel garbage collection. The heap sizes of these benchmarks were set to 3x of their respective minimum heap sizes [79, 159]. Details on heap configuration are listed in Table 6.2. To minimize non-determinism in multicore environments, each result was the average of 10 runs.

- DaCapo is an open source client-side Java benchmark suite which consists of a set of real-world applications with non-trivial memory loads. The version of DaCapo we used in this work is 9.12.
- SPECjvm2008 is a benchmark suite for measuring the performance of a Java Runtime Environment, containing several real life applications focusing on core Java functionalities. The workloads mimic a variety of common general-purpose application computations.
- HiBench is a big data benchmark suite that contains a set of Hadoop, Spark and streaming workloads, including kmeans, wordcount, and pagerank, etc. The version we used in this work is 6.0. We ran HiBench on Hadoop [13] 2.7.3 and Spark [139] 2.0.0 on a single node. The number of threads in the Spark executor was set to 16.
- Cassandra is an open-source distributed database management system designed to handle large amounts of data across many commodity servers. We use cassandra-
stress to test the read and write latency. The version of Cassandra used in this work was 3.0.10.

6.5.2 Improvement in Overall Performance

We first demonstrate the effectiveness of our optimizations on improving the overall performance of various Java applications. We adopted five benchmarks from DaCapo and SPECjvm2008, respectively, and compared their performance in the vanilla HotSpot JVM with that in our optimized JVM. Figure 6.10 (a) shows the execution times of five benchmarks from DaCapo under various settings. To quantify the performance improvement due to each optimization, we enabled one optimization at a time, GC affinity or optimized stealing, while disabling the other. The resulted JVMs are labeled as w/ GC-affinity and w/ steal, respectively. Together refers to the JVM with both optimizations enabled. Overall, GC thread affinity contributed more improvement compared to optimized stealing. For instance, the execution time of sunflow was improved by 30.4% and 17.5% due to thread
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Suite</th>
<th>Heap size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>DaCapo</td>
<td>900</td>
</tr>
<tr>
<td>jython</td>
<td>DaCapo</td>
<td>90</td>
</tr>
<tr>
<td>lusearch</td>
<td>DaCapo</td>
<td>90</td>
</tr>
<tr>
<td>sunflow</td>
<td>DaCapo</td>
<td>210</td>
</tr>
<tr>
<td>xalan</td>
<td>DaCapo</td>
<td>150</td>
</tr>
<tr>
<td>compiler.compiler</td>
<td>SPECjvm</td>
<td>4000</td>
</tr>
<tr>
<td>compress</td>
<td>SPECjvm</td>
<td>2500</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>SPECjvm</td>
<td>2500</td>
</tr>
<tr>
<td>xml.transform</td>
<td>SPECjvm</td>
<td>4000</td>
</tr>
<tr>
<td>xml.validation</td>
<td>SPECjvm</td>
<td>4000</td>
</tr>
<tr>
<td>kmeans</td>
<td>HiBench</td>
<td>16384</td>
</tr>
<tr>
<td>wordcount</td>
<td>HiBench</td>
<td>16384</td>
</tr>
<tr>
<td>pagerank</td>
<td>HiBench</td>
<td>16384</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Apache</td>
<td>8192</td>
</tr>
</tbody>
</table>

Table 6.2: Benchmark heap size.

affinity and optimized stealing, respectively. The performance improvement was more pronounced when both optimizations were taking effect. *sunflow* had a performance gain of 37.3% with *together*. Figure 6.10 (b) shows the throughput of *SPECjvm2008* benchmarks. Similarly, each optimization accelerated the operation speed of *SPECjvm2008* benchmarks. For instance, the throughput of *xml.transform* was improved by 18.9% and 10.3% with the thread affinity and optimized stealing, respectively. The combined optimizations improved its throughput by 24.3%. Note that the overall performance gain is lower than the aggregate gain of individual optimizations because each optimization improved the load balance, leaving less room to the other optimization for additional improvement.

Thread pinning has been used in HotSpot for improving data placement on non-uniform memory access (NUMA) machines. In particular, Gidra et. al. [68] proposed to segregate JVM heap in different NUMA nodes and restrict GC threads to only access objects in the local node. While the main purpose of segregated heap is to improve access balance across NUMA nodes, it imposes node affinity on GC threads, which could alleviate the GC thread stacking problem. In addition, segregated heap restricts work-stealing to GC threads within the same node, reducing the number of failed steal attempts before a GC thread enters the termination procedure. We ported the node affinity and NUMA-aware stealing proposed in [68] to OpenJDK 1.8.0 and compared their performance with our proposed optimizations.

Figure 6.11 (a) compares the performance of node affinity and our proposed dynamic thread affinity, using the vanilla JVM as the baseline. The results show that node affinity improved the overall performance compared to the vanilla JVM. The even distribution of
GC threads to the two NUMA nodes in our testbed mitigated thread stacking, thereby improving concurrency during GC. However, our proposed thread affinity still outperformed the node affinity in most DaCapo benchmarks while achieving comparable performance in the SPECjvm2008 benchmarks. It suggests that thread stacking can still happen within one NUMA node and the one-to-one thread-to-core binding is necessary for fully exploiting the potential of parallel GC.

Figure 6.11 (b) shows the results due to NUMA-aware stealing and our proposed adaptive stealing algorithm. From the figure, we can see that our approach achieved slightly better performance over NUMA-aware stealing. Note that we did not port NUMA-aware memory allocation proposed in [68]. The performance gain of NUMA-aware stealing was mainly due to reduced failed steal attempts. Since stealing is only allowed within a node, a GC thread stops stealing after $2 \times N_{local}$ failed attempts, where $N_{local}$ is the number of GC threads in a node. However, node affinity and NUMA-aware stealing share a common drawback – the static binding of GC threads to NUMA nodes and the restriction on where to steal GC tasks make the two approaches vulnerable to interference in a multi-user environment. In contrast, our approaches bind threads to the lightly loaded cores and steal from active threads, thereby more resilient to interference.

### 6.5.3 Improvement on Garbage Collection

Figure 6.10 (c) shows the performance improvement on garbage collection in DaCapo and SPECjvm2008 benchmarks. Both optimizations were enabled and labeled as optimized-JVM. From this figure, we can see that our optimizations benefited all benchmarks and reduced GC time. The improvement in GC time ranged from 20% (compiler.compiler) to 87.1% (sunflow). In general, the benchmarks that did not have as much improvement as others did were those that had relatively low steal failure rate in Table 6.1, i.e., compiler.compiler and xml.validation. The low steal failure rate indicated good balance among GC threads, thereby less room for GC improvement. It confirms that load imbalance is the major inefficiency in GC on these benchmarks.

### 6.5.4 Scalability

GC load varies depending on the activities of mutator threads on the heap. Therefore, it is interesting to study how the number of mutators affects GC scalability and the performance impact of GC on overall application scalability. We let Parallel Scavenge configure the number of GC threads (i.e., 15 GC threads) and varied the number of mutators from 1 to 16. We evaluated two types of applications in DaCapo: non-scalable and scalable benchmarks [125]. As shown in Figure 6.12, there is not much parallelism in non-scalable applications, such as h2 and jython, and performance deteriorated or stagnated as the
Figure 6.12: DaCapo: overall and GC scalability. (mutator varies from 1 to 16, GC thread number is 15)

number of mutators increased. In contrast, for scalable applications, such as lusearch, sunflow and xalan, performance improved with more mutators but became non-scalable as mutator concurrency continued to increase. We had three observations: 1) GC overhead depends on mutator activities. For non-scalable applications, more mutators did not exploit much parallelism, thereby not creating more activities on the heap. GC time was relatively stable with a varying number of mutators. For scalable applications, GC overhead increased with mutator activities as load imbalance could arise. The increased GC time could possibly affect application scalability. For example, the GC time jump in lusearch from 4 to 8 mutators was the inflection point of its scalability curve. 2) The optimized JVM significantly reduced GC time in all applications and was not sensitive to mutator activity. Hence, it does not affect application scalability. 3) The improved load balance in GC also helped improve mutator performance as the overall performance gain in all applications was greater than
the savings on GC time. An explanation is that good load balance during GC keeps multiple cores active. When the STW pause ends, waking mutator threads will have lower startup latency on the active cores compared to waking up from idle cores in deep power saving states. Further, active cores perform more frequent load balancing and help prevent mutator imbalance.

6.5.5 Application Results

Big data applications. We used wordcount, pagerank and kmeans from HiBench to evaluate the effectiveness of our design in a Spark environment. We set the heap size of Spark executors to 16GB and mutator thread to 16. Figure 6.13 (a) shows the performance of these applications with three pre-defined data sizes: small, large and huge. Note that pagerank with the huge dataset crashed due to out-of-memory errors and was not included in the figure. The results show that the optimized JVM consistently outperformed the vanilla JVM. The performance improvement on individual applications increased as the datasets became larger. However, the overall performance improvement of big data applications was not as great as DaCapo or SPECjvm2008 benchmarks. For example, the greatest
performance gain was 15.3% observed in kmeans with the huge dataset. The improvement was from reduced GC and mutator time. In kmeans, the GC time accounted for 50.3% of the total execution time, about two thirds of which was due to full GC. It has been reported that GC on the old generation is the bottleneck in big data application due to the caching of Resilient Distributed Dataset (RDD) [116]. Thus, scanning such a huge object in the heap is different from other GC tasks. The optimized JVM mainly reduced the time in minor GC for big data applications.

**Database applications.** Figure 6.13 (b) and (c) show the latency of read and write requests in Cassandra database. Cassandra was launched in a JVM with a 8GB heap and the client machine remotely sent requests using 256 threads. The concurrency setting on the client achieved the maximum Cassandra throughput. The results show that our optimized JVM was most effective on reducing the tail latency while had marginal improvement on the mean and median latency. It improved the 99th percentile read latency in Cassandra by 43%. The improvement on the tail latency led to up to 31% increase on Cassandra throughput (not shown in the figure). In what follows, we show more detailed statistics on GC time of these applications.
**Application GC time.** Figure 6.13 (d) shows the normalized GC time in Spark and Cassandra database. As discussed earlier, the saved GC time in the optimized JVM contributed most to overall performance improvement of big data applications. Compared to the small data size, the optimizations achieve more improvement of GC on the large data size. For database operations, both the total and average GC time in Cassandra are reduced in the optimized JVM. The optimized GC time was only about half of that in the vanilla GC. The reduction on the STW pause helped rein tail latency.

### 6.5.6 Results on Different Heap Sizes

For Java applications, it is important to set an appropriate heap size to prevent too frequent garbage collection while minimizing memory usage. This section studies the performance of the optimized JVM with different heap sizes. The smaller the heap size, more frequently would GC be performed but each GC takes less time as the space to scan is smaller. On the other hand, a larger heap requires GC to scan more space and each GC takes longer. Figure 6.14 shows the total execution time and GC time of *lusearch* and *kmeans* with different heap sizes. *lusearch* started with a minimum 30MB heap and increased the heap size at a step of 30MB until reaching 900MB. Both JVMs had improving performance as the heap size of *lusearch* increased. The GC time also continued to drop with larger heaps. At the minimum heap size, the performance gain due to our optimizations is narrower than that with larger heap sizes. The reason is that within such a small heap, the overhead of managing multiple GC threads may outweigh the benefit of concurrency. Improved load balance in the optimized JVM can hurt performance due to loss of locality. Nevertheless, the optimized JVM still outperformed the vanilla JVM by a large margin. The vanilla JVM can achieve comparable performance with the optimized JVM only with a much large memory footprint. Figure 6.14 (c) and (d) show a slightly different trend in *kmeans*. Instead of suffering poor GC performance with the minimum heap size in *lusearch*, the optimized JVM attained the most performance gain over vanilla JVM with 8GB minimum heap and had diminishing gain as heap size increased. As the proportion of the GC time in the total execution time decreases with larger heap size, performance improvement which derives from the optimization on GC time reduction diminishes as the GC time becomes less significant to the total time.

### 6.5.7 Results with Multiple Applications

We further evaluated our optimizations in a multi-application environment. First, we ran ten busy loops on ten cores to emulate some background workloads contending for the CPU. When *lusearch* is run along with the interfering workload, the OS load balancer tends to stack the GC threads onto a few cores. Static binding may also place GC threads
with the interfering loop on the same core. As Figure 6.15 (a) and (b) depict, dynamic GC thread balancing was able to reduce the total execution time and GC time of *lusearch* by 49.6% and 77.2% compared to the vanilla JVM. In addition, we evaluated the performance of multiple co-running JVMs. We executed two instances of *lusearch* or two instances of *sunflow* at the same time and the thread settings as well as the heap size were the same as above. Figure 6.15 (a) and (b) illustrate the improvement of total execution time and GC time of two *lusearch* or *sunflow*. When two JVM applications execute simultaneously, both the total and GC time increase compared to executing them alone in Figure 6.10. However, our optimizations still benefit both the overall and GC performance. This is due to the GC load balancing and smart stealing that help the applications perform more efficiently under constrained resources.

### 6.5.8 Discussion

**Simultaneous multithreading.** While existing studies have reported both constructive and destructive effects of SMT on application performance [99, 82, 130, 108], we found that enabling SMT can mitigate the thread stacking issue. For a fair comparison and to avoid over-threading in GC, we fixed the number of GC threads to 15 to match the number of GC threads automatically determined by HotSpot when SMT was disabled. Figure 6.16 shows that enabling SMT improved the overall performance. In CFS, the default interval between periodic load balancing is 64ms with SMT enabled and 128ms with SMT disabled. Thread migration between sibling hyperthreads is considered cheaper than that across physical cores and is thus performed more frequently. In addition, with SMT enabled, cores are less likely to enter a low-power state as activities on either of its two hyperthreads will prevent the core from idling. This will also increase the frequency of *idle_balance* in CFS. Nevertheless, SMT does not eliminate thread stacking and our approach further improves performance via thread affinity and adaptive stealing.
Beyond garbage collection. There exists an inherent tradeoff between optimizations on synchronization and OS scheduling. On the one hand, synchronization optimizations limit the number of concurrent lock contenders to reduce either futile park-unpark activity or CAS contention. On the other hand, OS load balancing is effective only if threads that are to be balanced are visible to the OS scheduler. Blocked threads are not eligible for load balancing as they do not contribute to the load. Besides parallel GC, we also observed a similar thread stacking issue in the futex-wake benchmark from perf benchmarks. The common issue is that programs with fine-grained blocking synchronization suffer execution serialization because the OS load balancing is ineffective. To address this issue, the OS could choose to balance blocked threads or rely on applications to provide hints on how to distribute threads on cores.

6.6 Conclusion

In this work, we identified vulnerabilities in the HotSpot JVM and Parallel Scavenge that can inflict loss of concurrency in parallel GC. We performed an in-depth analysis of the issue and revealed that it resulted from complex interplays among dynamic GC task assignment, unfair mutex locking, imperfect OS load balancing and less efficient stealing during the GC. We proposed an effective approach which coordinated the JVM with the Operating System to address GC load imbalance and designed a more efficient work stealing algorithm. Experiment results showed consistent performance improvement compared to the vanilla HotSpot JVM in various types of applications.
CHAPTER 7
CONCLUSIONS AND FUTURE WORK

Cloud computing has become the fundamental technology to support numerous applications and businesses due to its elastic resource allocation, operation cost reduction as well as service availability and productivity enhancement. Meanwhile, many inherent imperfection of the cloud, such as additional virtualization overhead, scheduling delays as well as semantic gaps among hardware, operating system and applications, have non-negligible impact upon the performance and quality-of-service of the cloud applications, especially for the I/O-intensive services. The goal of this dissertation lies in simple yet effective solutions that characterize and improve the network performance in different virtualized systems of the cloud. From the perspective of characterization, this dissertation covers network activities tracing on VM scheduling, container network bandwidth, virtualized network devices, etc. From the perspective of optimization, this dissertation proposes solutions which coordinate hypervisor, operating systems and application runtime such as JVM to improve the network performance. In the rest of this chapter, I will summarize the contributions of this dissertation and discuss the opportunities of future work.

7.1 Contributions
The contributions of this dissertation can be summarized as follows:

- We designed an in-band profiler named *Time Capsule* to provide fine-grained and across boundaries latency tracing in virtualized networks with acceptable overhead.
- We built *vNetTracer*, an efficient and programmable packet profiler for virtualized networks. It leverages extended Berkeley Packet Filter to add user-defined trace programs into virtualized networks to enable lightweight and dynamic tracing.
- We identified the priority inversions in virtualized OSes during discontinuous time and proposed *xBalloon*, a lightweight approach to preserving I/O prioritization between I/O- and compute-bound programs.
- We presented an in-depth performance analysis of GC in HotSpot JVM and proposed optimizations which enforces GC thread affinity to aid multicore load balancing and accelerates work stealing.
7.2 Future Work

In my future research, I will continue my effort on the improvement of application performance and system efficiency in virtualized network systems. In the near term, I plan to further explore more advanced solutions which integrated the hardware, hypervisor, operating systems and applications to deliver better virtualized network performance and higher system utilization. Specifically, I will devote myself in characterizing the lightweight virtualized systems, and improving the underlying cloud infrastructure, especially containers [74, 81, 95], for better Quality-of-Service, more predictable performance and better fairness guarantee. In the long term, I will aim to expand the current experience in improving the virtualized network into the new forms of services and technologies. I will explore the network systems in edge computing or IoT devices, and investigate the possibilities of integrating machine learning with the network optimization.

7.2.1 Low Overhead Profiling for Light Cloud Networks

Network performance problems or bottlenecks are notoriously tricky to diagnose, and this is magnified when applications are running on top of the increasing complex and consolidated virtualized network in the data centers. However, it is necessary to instrument and monitor the system and the applications in order to guarantee its normal execution and trouble shooting. This semantic gap raises the requirement of a highly efficient and low overhead network profiling for services in the cloud environment. In *Time Capsule* and *vNetTracer* project, I have proposed a few solutions for addressing the challenges of tracing the cloud networks. In recent years, many new techniques, such as containers, Kubernetes, microservices, etc., have been widely adopted in the data centers. These new frameworks and services make the resources in the cloud more fine-grained, increase the scale of cloud applications and introduce more fragmented organization and complex architecture. All the above characteristics require new and advanced solutions to monitor their performance and identify their issues in these new forms of cloud networks. Therefore, I plan to extend my current research into these new areas and explore the efficient solutions for profiling and monitoring services in the latest cloud environment.

7.2.2 High Performance Cloud Network

Network performance is always one of the top concerns for both service customers and cloud providers. As the cloud economy increases and the cloud ecology grows, both horizontal and vertical scope of cloud are expanding. On one hand, the scale of cloud keeps increasing with more servers involved, more data centers built and more customers connected. On the other hand, the hierarchy of network as well as the software stack becomes much thicker and more complicated compared to the early era of cloud. All of the above
make the network less efficient and more unpredictable for the cloud services. In xBalloon and Optimized JVM GC project, I have identified and addressed a few critical network performance bottlenecks in VMs, containers and application runtime. The ultimate goal of my research is to mitigate or even eliminate the abstraction overhead of virtualization in different layers inside the entire systems. Based on my previous experiences in building systems, application profiling and problem solving, I will continue leveraging my expertise to carry out influential work for this goal.

7.2.3 Optimizing Network Performance in Edge Computing

As the emerging applications and services such as 5G networks, Internet of Things (IoT), serverless architecture, embedded artificial intelligence appear, the traditional cloud is becoming increasingly inadequate to support such new technologies. To meet the latest requirements including real-time analytics, local control, constrained network bandwidth, data security and so forth, edge computing has been proposed in recent years by integrating the local computing, storage, networking and resource management together with the applications. With such a design, the edge cloud can effectively address the challenges that the traditional cloud cannot handle. Similar with the development of cloud in the past ten years, there exist massive business opportunities as well as research challenges in the future edge computing. For example, academia experts should rethink the traditional network designs and reshape the network architectures to meet the tomorrow cloud. As a network and system researcher, I am interested in studying the networking infrastructure challenges inside the edge computing and exploring the potential solutions to meet the growing demands.

7.2.4 Machine Learning in Network Monitoring and Optimization

As the scale and the complexity of cloud grow, it is becoming increasingly struggling to effectively monitor and manage the immense system network, let alone providing comprehensive and actionable insights. On the other hand, there exist massive inefficient parts inside the whole system. Misconfigurations, failures, security attacks interfere the execution of the cloud network all the time. In recent years, the emergence of machine learning and artificial intelligence (AI) promise new opportunities to tackle the above challenges in understanding and managing cloud networks. In my future research, I am interested in exploring the possibilities of integrating machine learning and AI with the network optimization. For instance, questions such as "What are good target problems in networking systems that might be solved by machine learning?", "What developments in artificial intelligence can impact networking optimization?", "How to balance the human
and machine role in managing the cloud networks?" are waited to investigate and I plan to answer these questions in my future research.
REFERENCES


VCPU which is not put back to active VCPU list in time will cause unpredictable long tail latency. http://lists.xenproject.org/archives/html/xen-devel/2016-05/msg01362.html.


