High-Throughput and Predictable VM Scheduling for High-Density Workloads

Thesis approved by
the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Manohar Vanga

Date of Defense:	26.11.2020
Dean:	Prof. Dr.-Ing. Jens B. Schmitt
Reviewer:	Björn Brandenburg
Reviewer:	Paul Francis
Reviewer:	Gabriel Parmer

D 386

ABSTRACT

In the increasingly competitive public-cloud marketplace, improving the efficiency of data centers is a major concern. One way to improve efficiency is to consolidate as many VMs onto as few physical cores as possible, provided that performance expectations are not violated. However, as a prerequisite for increased VM densities, the hypervisor's VM scheduler must allocate processor time efficiently and in a timely fashion. As we show in this thesis, contemporary VM schedulers leave substantial room for improvements in both regards when facing challenging high-VM-density workloads that frequently trigger the VM scheduler. As root causes, we identify (i) high runtime overheads and (ii) unpredictable scheduling heuristics.

To better support high VM densities, we propose Tableau, a VM scheduler that guarantees a minimum processor share and a maximum bound on scheduling delay for every VM in the system. Tableau combines a low-overhead, core-local, table-driven dispatcher with a fast on-demand table-generation procedure (triggered on VM creation/teardown) that employs scheduling techniques typically used in hard real-time systems. Further, we show that, owing to its focus on efficiency and scalability, Tableau provides comparable or better throughput than existing Xen schedulers in dedicated-core scenarios as are commonly employed in public clouds today.

Tableau also extends this design by providing the ability to use idle cycles in the system to perform low-priority background work, without affecting the performance of primary VMs, a common requirement in public clouds.

Finally, VM churn and workload variations in multi-tenant public clouds result in changing interference patterns at runtime, resulting in performance variation. In particular, variation in *last-level cache* (LLC) interference has been shown to have a significant impact on virtualized application performance in cloud environments [124]. Tableau employs a novel technique for dealing with dynamically changing interference, which involves periodically regenerating tables with the same guarantees on utilization and scheduling latency for all VMs in the system, but having different LLC interference characteristics. We present two strategies to mitigate LLC interference: a randomized approach, and one that uses performance counters to detect VMs running cache-intensive workloads and selectively mitigate interference.

Acknowledgements

This thesis was only made possible through the guidance and support of many advisors, colleagues, collaborators, support staff, friends, and family.

I wish to first express my sincerest gratitude to my advisor, Björn Brandenburg. Thanks for taking a chance on me, for your advice over the last seven years, for your persistent encouragement and patience, and for providing me with the learning experience of a lifetime. The lessons I learnt from you, I will carry with me for the rest of my life.

I also wish to thank other mentors and advisors who have helped me along the way. Paul Francis and Gabriel Parmer for being on my thesis committee, and for providing me with invaluable feedback on my research. Sameh Elnikety and Ricardo Bianchini for their mentorship during my internship at Microsoft Research. A big thanks to my collaborators and co-authors, Arpan Gujarati, Felipe Cerquieira, Andrea Bastoni, Henrik Theiling, Pratyush Patel, Roy Spliet, Mahircan Gül, Anna Lyons, and Gernot Heiser. Working with them was highlight of my studies and an incredible learning experience. I also wish to thank the other faculty members of the MPI-SWS SysNets Group, including Peter Druschel, Deepak Garg, Krishna Gummadi, Paul Francis, Allen Clement, Jonathan Mace, and Antoine Kaufmann. Interacting with them has been a privilege of a lifetime. I especially want to thank Rose Hoberman for her writing and presentation courses, both of which helped me immensely in improving my communication skills, as well as her invaluable feedback on early paper drafts during the course of my studies. Finally, I wish to thank the wonderful students who I got to work with in an advising capacity during the course of my doctoral studies: Cosmin Marin, Pratyush Patel, Mahircan Gül, Elena Lucherini, and Roy Spliet.

I want to especially thank Arpan and Felipe for their company during the countless all-nighters spent in the office getting our papers in shape for deadlines, the daily train rides between Saarbrücken and Kaiserslautern spanning five years, and the many conferences we attended together. Some of my most memorable moments from my Ph.D. are thanks to them.

I also want to thank the wonderful students that are or have been a part of the Real-Time Systems Group at MPI-SWS over the years. In no particular order: Arpan Gujarati, Alexander Wieder, Felipe Cerquiera, Mitra Nasri, James Robb, Tobias Blaß, Marco Perroni, Marco Maida, Elena Lucherini, Roy Spliet, Darshit Shah, Cosmin Marin, Mahircan Gül, Rohith Ramakrishnan, Felix Stutz, and Sergey Bozhko. They contributed

greatly towards the vibrant intellectual environment in our group during my time at MPI-SWS.

This thesis would not be possible without the help of the amazing administrative staff at MPI-SWS. I wish to thank Claudia Richter, Vera Schreiber, Susanne Girard, Annika Meiser, Mouna Litz, Gretchen Gravelle, Roslyn Stricker, Maria-Louise Albrecht, and Corinna Kopke. If it were not for their help with the hundreds of administrative issues across the better part of a decade, life in Germany would have been significantly worse. In particular, I want to thank Claudia and Vera for patiently helping me through some of the more ridiculous administrative situations I got myself into over the years. Finally, I want to thank Tobias Kaufmann, Christian Klein, Carina Schmitt, Christian Mickler, and Pascal Briehl for their technical support. Quite literally, this thesis would not have been possible without their help. In particular, I want to thank Tobias for always responding kindly to my numerous last-minute requests to extend my cluster reservation, in order to prevent my experimental data from being wiped out by a heartless, unforgiving, automated script.

I am thankful to all the current and former members of MPI-SWS, especially Felipe Cerquieira, Alexander Wieder, Ezgi Cicek, Lisette Espin, Paarijat Aditya, Scott Kilpatrik, Utkarsh Upadhyay, Bilal Zafar, Ivan Gavran, Natacha Crooks, Eslam Elnikety, Ekin Istemi Akkus, Pramod Bhatotia, Bimal Vishwanath, Anne-Kathrin Schmuck, Jan-Oliver Kaiser, Oana Goga, Nancy Estrada, Lily Tsai, Heiko Becker, Debasmita Lohar, Abir De, Darshit Shah, and Pratyush Patel. I want to especially thank Reinhard Munz, Georg Neis, Aastha Mehta, Anjo Vahldiek-Oberwagner, and Alexander Wieder for their incredible friendship over the years. Finally, I want to give a special thanks to Juhi Kulshreshta, Prabhav Kalaghatgi, and Mittul Singh, who were my family away from home, and a constant source of encouragement and support during my time at MPI-SWS.

I am eternally grateful to my family: my mother and father who are the most loving and supporting parents anyone could ever ask for, and my sister whose humor kept me sane through it all. Last but certainly not least, I could not have done any of this without the incredible love and support of my wife Saakshita, who always believed in me and never gave up on me.

List of Publications

The work in this thesis is primarily based on the following publication:

- **Manohar Vanga**, Arpan Gujarati, and Björn Brandenburg, "Tableau: A High-Throughput and Predictable VM Scheduler for High-Density Workloads", Proceedings of the 13th European Conference on Computer Systems (EuroSys 2018)

Other publications published during the course of my doctoral studies, in the broader area of real-time systems, are listed below:

- **Manohar Vanga**, Andrea Bastoni, Henrik Theiling, Björn Brandenburg, "Supporting Low-Latency, Low-Criticality Tasks in a Certified Mixed-Criticality OS", Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS 2017)
- Pratyush Patel, **Manohar Vanga**, Björn Brandenburg, "TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference", Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2017) **Best Paper Award.**
- Roy Spliet, **Manohar Vanga**, Björn Brandenburg, Sven Dziadek, "Fast on Average, Predictable in the Worst Case: Exploring Real-Time Futexes in LITMUS^{RT}", Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS 2014)
- Felipe Cerqueira, **Manohar Vanga**, Björn Brandenburg, "Scaling Global Scheduling with Message Passing", Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014)

Curriculum Vitae

Education

2014 — 2020	Ph.D. in Computer Science (dissertation phase) *MPI-SWS & TU Kaiserslautern, Germany*
2012 — 2014	Ph.D. in Computer Science (preparatory phase) *MPI-SWS & Saarland University, Germany*
2012 — 2019	Master in Computer Science *Saarland University, Germany*
2006 — 2010	B.Tech. in Computer Science and Engineering *Jawaharlal Nehru Technological University, India*

Work Experience

Jul 2018 — Oct 2018	Research Intern, *Microsoft Research, USA*
Jan 2011 — Jun 2012	Project Associate, *CERN, Switzerland*
Dec 2009 — Apr 2010	Intern, *University of Pavia, Italy*

Awards & Professional Activities

Best Paper Award	*RTAS 2017*
External Reviewer	EuroSys *(2013, 2016, 2019)*, RTSS *(2013, 2016, 2018)*, RTAS *(2013, 2014, 2017)*, ECRTS *(2013—2015, 2019)*, RTNS *(2014—2016)*, EMSOFT *(2015—2016)*, SYSTOR *(2015—2016)*, Middleware *(2018)*

Teaching and Advising

2018, Advising	Cosmin Marin, Master Thesis *MPI-SWS*
2016, Advising	Pratyush Patel, Summer Intern *MPI-SWS*
2014, Teaching Assistant	Distributed Systems, Winter Semester *MPI-SWS & Saarland University*
2013, Teaching Assistant	OS Reading Group, Summer Semester *MPI-SWS & Saarland University*

Table of Contents

Abstract . . . ii

Acknowledgements . . . ii

List of Publications . . . v

Curriculum Vitae . . . vii

Table of Contents . . . ix

List of Figures . . . xiii

List of Tables . . . xvii

1 Introduction . . . 1

2 Background and Prior Work . . . 7

2.1 Cloud Computing and OS Virtualization . . . 7

2.1.1 Cloud Computing Terminology . . . 8

2.1.2 OS Virtualization . . . 9

2.1.3 Intel VT-x and AMD-V . . . 12

2.1.4 Types of Hypervisors . . . 13

2.1.5 VMs vs. Containers . . . 14

2.1.6 VMs and Unikernels . . . 15

2.2 The Xen Hypervisor . . . 16

2.3 VM Scheduling in Xen . . . 17

2.3.1 The Xen Scheduler Framework . . . 18

2.3.2 The Credit Scheduler . . . 20

2.3.3 RTDS Scheduler . . . 21

2.4 VM Scheduling in Other Hypervisors . . . 23

2.5 Real-Time Scheduling . . . 25

2.5.1 Periodic Task Model . . . 25

2.5.2 Schedulability, Feasability, and Optimality . . . 25

2.5.3 Partitioned Earliest-Deadline-First Scheduling . . . 26

2.5.4 Space and Time Partitioning with ARINC 653 . . . 27

2.6 Performance Monitoring on Intel Platforms . . . 29

2.7 Prior Work . . . 29

3 The Tableau VM Scheduler . . . 39

3.1 The Role of a VM Scheduler . . . 39

3.1.1 Heuristics Increase Tail Latencies . . . 40

3.1.2 Scheduling Overheads Limit Throughput . . . 41

3.2 Design Overview . . . 42

3.2.1 Dispatcher vs. Planner . . . 42

3.2.2 Second-Level Scheduler . . . 43

3.2.3 Background Scheduler . . . 44

4 DISPATCHER IMPLEMENTATION . . . 47
4.1 Table-Driven Dispatcher . . . 47
4.2 Tier-1 Scheduling . . . 49
4.2.1 Table-Driven Scheduler . . . 49
4.2.2 Second-Level Scheduler . . . 52
4.3 Tier-2 Scheduling . . . 53
4.4 Summary . . . 55

5 PLANNER DESIGN . . . 57
5.1 Mapping vCPUs to Periodic Tasks . . . 58
5.2 Partitioning . . . 61
5.3 Semi-Partitioning . . . 62
5.4 Localized Optimal Scheduling . . . 64
5.5 Table Generation . . . 64
5.5.1 Post-Processing Tables . . . 65
5.5.2 Storing and Pushing Tables . . . 65
5.6 Summary . . . 65

6 EXPERIMENTAL RESULTS . . . 67
6.1 Table-Generation Overheads . . . 67
6.2 Scheduler Runtime Overheads . . . 70
6.3 Comparing Scheduling Delay . . . 73
6.4 Comparing `nginx` HTTPS Throughput (High Density) . . . 77
6.5 Comparing `nginx` HTTPS Throughput (Dedicated) . . . 85
6.6 Tier-2 Performance and Impact . . . 88
6.7 Discussion and Limitations . . . 92

7 MITIGATING DYNAMIC CACHE INTERFERENCE . . . 95
7.1 Motivation . . . 96
7.2 Mitigation Strategy 1: Random Repartitioning . . . 99
7.3 Mitigation Strategy 2: Load Balancing Based on Performance Counters . . . 102

8 CONCLUSION . . . 113
8.1 Summary of Results . . . 113
8.2 Open Questions and Future Work . . . 114

A EXTENDED EVALUATION: EFFECT OF VARYING SCHEDULING LATENCY . . . 117
A.1 Credit Scheduler . . . 118
A.1.1 Idle Background Workload . . . 118
A.1.2 Cache-Intensive Background Workload . . . 119
A.1.3 I/O-Intensive Background Workload . . . 121
A.2 Tableau Scheduler . . . 123
A.2.1 Idle Background Workload . . . 123
A.2.2 Cache-Intensive Background Workload . . . 124
A.2.3 I/O-Intensive Background Workload . . . 126
A.3 RTDS Scheduler . . . 128
A.3.1 Idle Background Workload . . . 128
A.3.2 Cache-Intensive Background Workload . . . 129
A.3.3 I/O-Intensive Background Workload . . . 130

B Extended Evaluation: Scheduling Latency Across Schedulers . 131
B.1 1ms Scheduling Latency . . . 132
B.1.1 Idle Background Workload . . . 132
B.1.2 Cache-Intensive Background Workload . . . 133
B.1.3 I/O-Intensive Background Workload . . . 135
B.2 5ms Scheduling Latency . . . 137
B.2.1 Idle Background Workload . . . 137
B.2.2 Cache-Intensive Background Workload . . . 138
B.2.3 I/O-Intensive Background Workload . . . 140
B.3 10ms Scheduling Latency . . . 142
B.3.1 Idle Background Workload . . . 142
B.3.2 Cache-Intensive Background Workload . . . 143
B.3.3 I/O-Intensive Background Workload . . . 145
B.4 15ms Scheduling Latency . . . 147
B.4.1 Idle Background Workload . . . 147
B.4.2 Cache-Intensive Background Workload . . . 148
B.4.3 I/O-Intensive Background Workload . . . 150
B.5 20ms Scheduling Latency . . . 152
B.5.1 Idle Background Workload . . . 152
B.5.2 Cache-Intensive Background Workload . . . 153
B.5.3 I/O-Intensive Background Workload . . . 155
B.6 25ms Scheduling Latency . . . 157
B.6.1 Idle Background Workload . . . 157
B.6.2 Cache-Intensive Background Workload . . . 158
B.6.3 I/O-Intensive Background Workload . . . 160
B.7 30ms Scheduling Latency . . . 162
B.7.1 Idle Background Workload . . . 162
B.7.2 Cache-Intensive Background Workload . . . 163
B.7.3 I/O-Intensive Background Workload . . . 165
B.8 35ms Scheduling Latency . . . 167
B.8.1 Idle Background Workload . . . 167
B.8.2 Cache-Intensive Background Workload . . . 168
B.8.3 I/O-Intensive Background Workload . . . 170

C Extended Evaluation: Dedicated-Core Performance Comparison 173
C.1 Idle Background Workload . . . 174
C.2 CPU-Intensive Background Workload . . . 175
C.3 Cache-Intensive Background Workload . . . 176
C.4 I/O-Intensive Background Workload . . . 177

D Tableau Table Generation Script . . . 179

Bibliography . . . 189

List of Figures

FIGURE 2.1: The two types of hypervisors 13
FIGURE 2.2: A comparison between virtual machines and containers 14
FIGURE 2.3: An overview of the PikeOS scheduler. The system comprises multiple application time partitions (TP_X), for which a static schedule is generated, and the system time partition (TP_0), which is always eligible (left). Each time partition consists of 256 ready queues and an associated bitmap tracking which levels have eligible tasks (middle). At runtime, the first task from the highest-priority, non-empty ready queue, in either TP_0 or the currently active TP_X, is dispatched (right). 27

FIGURE 4.1: Tableau architecture 48
FIGURE 4.2: Scheduling table and slice table 49

FIGURE 5.1: Example scheduling table for the semi-partitioned task set given in Table 5.3 63

FIGURE 6.1: Table-generation times for a varying number of VMs with different latency goals. The 30 ms and 100 ms curves overlap. 69
FIGURE 6.2: Generated table size for a varying number of VMs with different latency goals. All but the 1 ms curve overlap. 70
FIGURE 6.3: Maximum scheduling delay as measured by `redis-cli`. "BG" denotes background workload. 74
FIGURE 6.4: Average and maximum-observed round-trip ping latencies. "BG" denotes background workload. 76
FIGURE 6.5: Mean (first column), 99$^{\text{th}}$ percentile (second column), and maximum (third column) observed latency for capped (first three rows) and uncapped scenarios (last three rows), for 1 KiB, 100 KiB, and 1 MiB files (see Y-axis labels), with an I/O-intensive background workload and with varying throughput. 79
FIGURE 6.6: Mean (first column), 99$^{\text{th}}$ percentile (second column), and maximum (third column) observed latency for capped (first row) and uncapped VMs (second row) for 100 KiB files with a cache-thrashing background workload and varying throughput. 81
FIGURE 6.7: Mean (first column), 99$^{\text{th}}$ percentile (second column), and maximum (third column) observed latency for a capped, high-density scenario for 1 KiB files, with three background workloads (I/O-intensive, CPU-bound cache-intensive, and idle) and with varying throughput. . . 83
FIGURE 6.8: Mean (first column), 99$^{\text{th}}$ percentile (second column), and maximum (third column) observed latency for an uncapped, high-density scenario for 1 KiB files, with two background workloads (I/O-intensive and CPU-bound cache-intensive) and with varying throughput. Results with the idle background workload are omitted as they are equivalent to a dedicated core scenario, and therefore not comparable. 84

FIGURE 6.9: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency with a dedicated-core scenario for 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying throughput. 86
FIGURE 6.10:Mean (first column), 99th percentile (second column), and maximum (third column) observed latency with a dedicated-core scenario for 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying throughput. 88
FIGURE 6.11:Mean (first column), 99th percentile (second column), and maximum (third column) observed latency with a dedicated-core scenario, and a tier-3 VM running an interfering workload, for 1 KiB, 100 KiB, and 1 MiB files and with varying throughput. 89
FIGURE 6.12:Mean (first column), 99th percentile (second column), and maximum (third column) observed latency of tier-3 VM co-located with a tier-1 VM capped at varying utilizations (20%, 40%, 60%, and 80%) and running a CPU-intensive workload, for 1 KiB files and with varying throughput. 90
FIGURE 6.13:99th percentile latency of a tier-1 VM capped at varying utilizations (20%, 40%, 60%, and 80%) and co-located with a tier-2 VM running various background workloads (idle, CPU-intensive, cache-intensive, and I/O-intensive), for 1 KiB files and with varying throughput. 91

FIGURE 7.1: A comparison of `canneal` performance running in a VM in a high-density scenario under varying number of interfering VMs running two types of interference workloads. 98
FIGURE 7.2: A comparison of `canneal` performance running in a VM under a no-interference scenario (Baseline), with thirty CI-1 VMs, thirty CI-2 VMs, and thirty CI-2 VMs with randomized mitigation 100
FIGURE 7.3: A comparison of `canneal` performance running in a VM in a high-density scenario under varying number of interfering VMs in two scenarios: one with random CI-1 interference, and one with CI-2 interference but our randomizer active. 101
FIGURE 7.4: Trend lines tracking LLC misses for a single VM and varying time windows. The VM goes through three phases of sixty seconds each: idleness, CPU-intensive work, and again idleness. 105
FIGURE 7.5: Trend lines tracking LLC misses for a single VM and varying time windows. The VM goes through three phases of sixty seconds each: idleness, CPU-intensive work, and again idleness. 106
FIGURE 7.6: A comparison of CPI over a five second interval, measured for a VM when it is idle, when running a CPU-intensive loop, and when running a cache-intensive workload 106
FIGURE 7.7: Trend lines tracking number of cycles executed by a single VM and varying time windows. The VM goes through three phases of sixty seconds each: idleness, CPU-intensive work, and again idleness. 107
FIGURE 7.8: A comparison of `canneal` performance running in a VM under **(i)** a no-interference scenario (Baseline), **(ii)** with all VMs on a socket causing cache interference, and **(iii)** with the same VMs spread across both sockets in the system. 109

FIGURE 7.9: A comparison of VM performance under a no-interference scenario (Baseline), with three interfering VMs are co-located on the same core as the evaluated VM, and when the three VMs are spread across cores of the same socket. 109
FIGURE 7.10:A comparison of VM performance under a no-interference scenario (Baseline), with thirty CI-1 VMs, thirty CI-2 VMs, thirty CI-2 VMs with performance-counter-based mitigation, and thirty CI-2 VMs with randomized mitigation 111

LIST OF TABLES

TABLE 5.1: Example vCPUs corresponding to three VMs, each with a utilization U and maximum scheduling latency L. 60
TABLE 5.2: The equivalent periodic tasks corresponding to the vCPUs shown in Table 5.1 with a budget C and a period T. 60
TABLE 5.3: Semi-partitioned example tasks 63

TABLE 6.1: Average runtime overheads (in μs) for three key scheduler-related operations on a 16-core, 2-socket server. 72
TABLE 6.2: Average runtime overheads (in μs) for three key scheduler-related operations on a 48-core, 4-socket server. 73

Dedicated to my family, especially my parents.

Chapter 1

Introduction

As the marketplace for public clouds matures and cloud services are being increasingly commoditized, cloud providers are forced to continuously increase the efficiency of their data centers, and to improve the price/performance ratio of their various service tiers, especially at the low end.

One way to increase data center efficiency is to pack a growing number of VMs onto fewer physical cores. This reduces resource wastage as active cores serving multiple lower-tier VMs are highly utilized while the number of paying customers relative to the required infrastructure can be increased. Alternatively, any freed-up cores can be used to support higher-tier (and higher-priced) VMs that require dedicated processing cores.

Regardless of how freed-up resources are utilized, the ability to consolidate a larger number of lower-tier VMs onto fewer cores—*i.e.*, the ability to pack VMs as tightly as possible without violating customer expectations—is a distinct economic advantage in the competitive cloud marketplace. However, consolidating VMs onto shared cores is easier said than done as customers desire high throughput and reasonably low and stable latency characteristics even for lower-tier VMs.

A key hypervisor component that affects these central metrics—application throughput and latency (as perceived by the customer)—is the *VM scheduler*. However, if the VM scheduler is a major bottleneck, then it will surely impose a tax on application performance. In particular, if the VM scheduler is inefficient (*i.e.*, if it suffers from large runtime overheads), then the peak throughput attainable by guest VMs will be needlessly limited. Furthermore, while application tail latency is a complex phenomenon that is determined by multiple factors, if the VM scheduler occasionally *induces* a substantial amount of *scheduling latency* due to poor scheduling decisions, then application tail latency will inevitably suffer.

Unfortunately, many of the VM schedulers in widespread use today are not yet optimized for hosting highly consolidated, high-VM-density workloads, and have little to offer in terms of performance *guarantees*. In particular, as we show later (Chapter 6) in an evaluation of the popular open-source Xen hypervisor, existing VM schedulers can negatively affect either tail latencies, throughput, or both due to the use of unpredictable scheduling heuristics that sometimes backfire and/or implementation aspects that cause undesirably high overheads, especially when faced with many densely packed VMs.

Further, VM schedulers are only one part of the puzzle to achieving high throughput and predictable latency in VMs in dynamic, multi-tenant public cloud environment. A secondary issue that arises is higher interference and performance variability due to increased micro-architectural resource contention, which must be addressed in order to achieve these goals. Packing VMs tightly onto cores exacerbates micro-architectural contention due to hardware-level resource sharing (*e.g.*, shared memory caches). This is particularly problematic due to the multi-tenant nature of public clouds, where workload characteristics of VMs co-located on the same machine vary over time, making it insufficient to simply ensure isolation at VM admission time. Rather, such contention needs to be dealt with continuously and isolation mechanisms need to be actively reconfigured at runtime accordingly.

This thesis. Motivated by these observations, and to better support highly consolidated VM workloads in public clouds, this thesis presents *Tableau*, a highly predictable, high-throughput VM scheduler based on an unorthodox design not previously explored in a data-center context. Specifically, Tableau leverages multiprocessor scheduling techniques typically used in hard real-time systems, and exploits specific properties of cloud environments to minimize runtime overheads, unlike prior real-time VM schedulers (*e.g.*, RT-Xen [120]). Tableau also supports other common requirements of cloud schedulers, namely the ability "to soak up" any idle cycles on a machine by running low-priority background VMs. Finally, Tableau also addresses the issue of cache-level contention through a novel load-balancing technique.

Tableau consists of two main components: **(i)** a low-overhead, table-driven, core-local dispatcher that schedules VMs primarily based on a given static schedule, and **(ii)** an asynchronous, infrequently invoked planner that re-generates tables on-demand when VMs are either created, torn down, or reconfigured.

As a result of this clear separation between a semi-offline planning phase and an extremely simple online dispatcher, Tableau incurs significantly lower runtime overheads (around $5.6\times$, $2.4\times$, and $2\times$ lower than under Credit, Credit2, and RTDS, respectively,

see Section 6.2). These efficiency gains in turn can translate into substantial improvements in SLA-aware throughput (*e.g.*, compared to RTDS, Tableau can achieve up to $1.6\times$ higher peak throughput when serving 1 KiB files with a 100 ms SLA, see Section 6.4).

Furthermore, Tableau's inherent predictability can yield substantially improved tail latency characteristics for workloads that frequently invoke the VM scheduler (*e.g.*, in some cases VMs scheduled by Tableau exhibit up to $17\times$ lower maximum ping latency compared to Credit, the most commonly used Xen scheduler, see Section 6.3).

Key to Tableau is the planning stage, which is performed asynchronously and only affects the creation, teardown, and reconfiguration time of VMs (inflating each one by a few hundred milliseconds). We consider this to be an acceptable trade-off for these relatively infrequent operations that usually take on the order of seconds to begin with.

Each VM under Tableau is configured with a minimum CPU budget (or *utilization*) and a maximum-acceptable scheduling delay, both of which can be determined either based on an explicit SLA, based on pre-determined, price-differentiated service tiers offered by cloud vendors, or empirically based on the deployed workload or simple fair-share policies. Tableau's planner applies techniques from hard real-time multiprocessor scheduling to *quickly* re-generate scheduling tables whenever needed, while ensuring that all constraints on the minimum utilization and maximum scheduling latency for every VM in the system are satisfied. Consequently, Tableau provides direct control over one of the key contributors to tail latencies, namely the scheduling latency of individual VMs.

A common requirement in public clouds is the ability to use any idle cycles in the system in order to perform low-priority background work, without affecting the performance of primary VMs (which are typically paid for by customers). The primary obstacle to achieving this is the lack of strong performance guarantees for VMs [46, 128], which Tableau provides. Subsequently, we present the design of a background scheduler that enables a lower-priority class of VMs (henceforth *tier-2 VMs*) to use any idle cycles in the system when no SLA-backed VMs (*i.e.*, table-driven VMs henceforth referred to as *tier-1 VMs*) are runnable. We present results that show that the impact of tier-2 VMs on the performance of tier-1 VMs is low, making it practical for cloud environments.

Finally, as there is VM churn in multi-tenant public clouds, and VMs on a single machine change workload characteristics, interference patterns change at runtime, resulting in performance variation. In particular, in this thesis, we focus on *last-level cache* (LLC) interference as it has been shown to have a significant impact on virtualized application performance in cloud environments [124]. We present a novel approach for

dealing with such changes in runtime interference, which involves periodically regenerating tables that provide the same guarantees on utilization and scheduling latency for all VMs in the system, but have different LLC interference characteristics.

We present two strategies to mitigate interference: a randomized approach, and one that uses performance counters to detect VMs running cache-intensive workloads and selectively mitigate interference. Both approaches employ a general mitigation strategy of distributing VMs causing LLC interference evenly across the cores in the system. The randomized approach does this by randomly shuffling the core assignments of all VMs in the system periodically, while the performance-counter-based version does this by first detecting "bad" VMs causing undue cache interference and selectively repartitioning them to distribute them evenly across the system. We present the design of both approaches, as well as compare their performance using a cache-sensitive subset of the PARSEC benchmark [15]. Our results show that randomizing tables works well for mitigating worst-case slowdowns due to cache interference. For the performance-counter-based approach, we explore different ways of detecting interfering VMs but conclude that a more robust detection mechanism is needed in order to match the performance of the randomized approach.

Contributions. This thesis is primarily based on the work presented in [108] and makes the following key contributions.

- We present the design of Tableau (Section 3.2), an unorthodox scheduling approach rooted in static scheduling tables (as pioneered in hard real-time systems [65]), which has not previously been explored in a cloud context. Tableau requires on-demand generation of scheduling tables satisfying the utilization and scheduling-latency constraints of individual VMs in the system.
- We detail how to quickly find such tables by repurposing relevant real-time scheduling theory (Chapter 5), and report on an efficient implementation of Tableau in Xen 4.9 (Chapter 4). Notably, our implementation is inherently scalable because it uses almost exclusively core-local data structures. In an evaluation with an I/O-intensive workload on a dual-socket, 16-core Intel Xeon platform (Chapter 6), our Tableau prototype is shown to outperform the existing Xen schedulers (RTDS, Credit, and Credit2) in terms of their SLA-aware peak throughput.
- We present the design and implementation of a background scheduler (Section 4.3) that enables a two-tier system for VMs, allowing for tier-1 VMs to have strong guarantees via the table-driven dispatcher, while tier-2 VMs execute on any spare idle cycles in the system with low impact on tier-1 performance (Section 6.6).

- We propose an extension of Tableau to address LLC interference from co-located VMs (Chapter 7). The primary challenge to doing so in a multi-tenant setting is dealing with the dynamic nature of such interference as workloads on co-located VMs, and consequently cache pressure, vary over time. We propose two approaches: a randomized one, and one that uses performance monitoring data for individual VMs. Both approach work by periodically regenerating Tableau's scheduling tables at runtime so as to lower peak interference on any single core.

CHAPTER 2

BACKGROUND AND PRIOR WORK

The contributions presented in this dissertation are built upon on a foundation of prior work, which we review briefly in this chapter. We also present the necessary background required to understand the contributions of this thesis.

This chapter is organized as follows. We begin by establishing basic terminology related to cloud computing (section 2.1) as well as providing basic background on OS virtualization technology (subsection 2.1.1). Next, we present a general overview of the Xen hypervisor (section 2.2), its scheduling framework in particular (section 2.3), and briefly discuss scheduling approaches used in some other popular commercial hypervisors (section 2.4). Next, we present core real-time scheduling background related to this thesis (section 2.5), followed by a brief overview of performance monitoring support on Intel platforms (section 2.6). Finally, we summarize prior work related to Tableau.

2.1 Cloud Computing and OS Virtualization

Cloud computing refers to delivering computing resources (compute, storage, and network) as a service, where they can be rented by customers on demand from a shared pool of physical compute resources, which are typically built as multiple specialized datacenters. Large players at the time of writing this thesis include Amazon's AWS, Microsoft's Azure, and Google's Cloud Platform.

The advantages of clouds are that **(i)** economies of scale for providers lowers costs for users, **(ii)** the pay-as-you-go pricing model allows customers to optimize costs by making resource usage *elastic* (*i.e.*, that is increase or lower resource availability on demand), and **(iii)** providers can intelligently multiplex customer workloads to improve overall

datacenter utilization. Finally, **(iv)** it lowers the total cost of ownership for businesses which no longer need to purchase, setup, and maintain their own datacenters.

2.1.1 Cloud Computing Terminology

We distinguish between three forms of cloud computing in this thesis: public, private, and hybrid clouds.

Public clouds are what "cloud computing" commonly refers to in the general usage of the term. In public clouds, users can quickly rent virtualized resources from providers, and are billed using a pay-as-you-go model where costs are incurred only for the amount of resource used. While there are advantages, as outlined above, there are also some disadvantages of using public clouds. Primarily, public clouds are unsuitable for businesses running applications that deal with highly sensitive information that must not be handled by third parties. Another common issue has to do with the need to comply with specific regulatory requirements (*e.g.*, HIPAA compliance for medical records, or PCI-DSS compliance for handling credit card information), although cloud providers are increasingly providing services tailored for dealing with such regulatory requirements as well [8].

Private clouds are at the other end of the cloud computing spectrum, where virtualized resources are only available to a single organization, typically implemented within a privately-owned data center. Private clouds offer strong data privacy due to the high degree of control organizations have over physical hardware, and can be tailored to specific regulatory requirements. However, these advantages come with a high initial investment cost, requiring dedicated maintenance personnel to ensure continued security, as well as limited elasticity in the face of changing load.

Finally, a *hybrid cloud* combines the best of both worlds by connecting a private and public cloud together. This allows the private section of a cloud to host critical or sensitive applications and data, while the non-critical workloads can be hosted on the public section. The only downside is the need to ensure a proper integration and connectivity between the two distinct cloud environments as the public component is likely to be maintained by a different organization. It should be noted that there is a difference between a private resource hosted within a public cloud and a private resource hosted in a private cloud. While both can be secured from public exposure to the internet by configuring internal networking rules, the former is under the control of the cloud provider, while the latter is entirely in control of the client organization. Therefore it is

not the safety with respect to public internet exposure that is at issue, but rather exposure to potentially untrustworthy cloud providers. Hybrid clouds provide the ability to maintain data safety both from the public internet and cloud organizations.

In this thesis, we *focus primarily on public clouds*, however, it should be noted that the techniques we describe are in no way limited to them. They are equally applicable in private and hybrid cloud settings.

2.1.2 OS Virtualization

The underlying technology that has enabled cloud computing is OS virtualization, which allows the multiplexing of multiple smaller *virtual machines* (VMs) on a larger physical machine, with each VM appearing as an independent, dedicated machine.

At the heart of virtualization is the *hypervisor*, a supervisory software whose primary function is to multiplex the physical resources on a given machine among multiple smaller VMs.

Intel x86 virtualization. Early attempts at virtualizing the Intel x86 architecture showed it to be difficult to virtualize for multiple reasons.

First, Intel processors provided no hardware provisions to aid virtualization. As a result, certain operations that only worked under privileged rings, and could not be executed in unprivileged rings, required emulation in software since guest OSes could not be run in privileged mode safely [106].

Second, various instructions demonstrated different behavior depending on which privilege ring one was operating in. For example, various instructions would fail silently when run in unprivileged rings [6] instead of causing an exception that would allow the hypervisor to trap and emulate it. This meant that running a hypervisor in the highest-privilege processor ring with OSes in lower-privilege rings would not always allow the hypervisor to observe invalid behavior by guest OSes. As a result, early virtualization techniques resorted to **(i)** trap-and-emulate techniques when possible [95], with problematic instructions being emulated at the cost of lowered performance, or **(ii)** to binary translation, where problematic instructions were re-written with trap instructions either statically at compile time or dynamically at runtime. These techniques result in lower performance compared to modern, hardware-assisted virtualization due to the additional work that needs to be done by the hypervisor at runtime.

Third, x86 virtualization was challenging as it also required many data structures of the guest OS to be *shadowed*. For example, OSes using virtual memory (most of them)

could not be safely granted access to the MMU as this would remove control from the hypervisor. Instead, a *shadow page table* would be provided to the guest that would trap memory access attempts, allowing the memory addresses being accessed to be translated in software.

Finally, virtualizing I/O required emulating the entire device in software, leading to a significant lowering in I/O performance.

To remedy this, Intel and AMD released the VMX and SVM instruction set extensions, respectively. The VMX and SVM instruction set extensions introduce a new privilege ring that has higher privileges than the four rings typically found in Intel x86 architectures (*i.e.*, VMX and SVM introduced ring -1 to the x86 architecture). Now, the hypervisor could be safely run in this new privileged ring, arbitrate hardware access for each VM in the system, while the OS within the VM would run in the familiar 4-ring x86 environment. Combined with *paravirtualization* (see below) to remove the need for I/O-device emulation, this removed the need for the heavyweight techniques described above in modern hypervisors.

We do not attempt to summarize the vast literature on virtualization techniques in this thesis. Rather, we refer the interested reader to a survey describing the historical evolution of virtualization techniques [26], as well as a survey of x86-specific virtualization techniques [6].

2.1.2.1 Types of Virtualization

We clarify various definitions for modern virtualization terminology by outlining the common types of virtualization referenced by hypervisor vendors and cloud providers at the time of writing this thesis.

Full software virtualization. In this type of virtualization, the entire hardware is emulated in software. The biggest advantage of full software virtualization is that any OS and application stack can potentially be hosted as the virtualization software is identical in its characteristics to the actual hardware the application was designed for.

This is advantageous for running legacy applications that have been designed at huge cost on new hardware (*e.g.*, safety-critical applications that have gone through expensive certification processes). It also does not require any specialized hardware with specific instruction set architectures or hardware modules. Any hardware that can run the emulation software can host the applications.

The big downside to full software virtualization is that it introduces high runtime overheads, which results in lower application performance and higher resource usage. For

example, in order to simulate a simple 6501 processor, every instruction in the 6501 instruction-set architecture must typically be emulated via multiple instructions of the host architecture. In addition, I/O access is slower since it must be transparently intermediated by emulated hardware, which adds additional runtime overheads compared to accessing the physical device directly.

Paravirtualization (PV). Under PV, the guest OS is modified to actively interface with the hypervisor. The guest OS therefore can choose to virtualize problematic parts of the system to improve performance. For example, rather than emulating an interrupt controller that would allow interrupt-based communication between the hypervisor and guest, the guest OS can implement an efficient mailbox-style system in shared memory to allow the hypervisor to "raise" interrupts by queueing messages into it.

Another advantage of PV is that I/O can be efficiently routed through a privileged OS that has privileged access to the underlying hardware, which obviates the need for re-implementing device drivers in the hypervisor. It also allows for complex resource-management policies to be implemented within the privileged OS as all I/O flows through it.

To summarize, the core idea behind PV is to make the guest aware of the fact that it is being virtualized, and have it actively interface with the hypervisor. Essentially, PV shows that if the cost of virtualizing the underlying architecture is too high or difficult to achieve, then one can virtualize an architecture closest to it that is virtualization friendly, and interface with the underlying hypervisor for the rest.

Hardware-assisted software virtualization (HV). Under HV, the hypervisor takes advantage of hardware features as described above (*e.g.*, Intel VT-x and AMD-V). Similar to software virtualization, some instructions are trapped and emulated, but this is done by specialized hardware that does not incur the same performance impact. This enables unmodified operating systems to be virtualized, without the downsides of full software virtualization (*i.e.*, lower performance due to emulation overheads).

Therefore, HV has the highest performance of all virtualization types as it makes use of specialized hardware features.

Typically, as we will see in the case of Xen, hardware-assisted virtualization is combined with paravirtualized device drivers in order to boost I/O performance. This is because I/O devices must still be shared, and the alternative to paravirtualization would be expensive emulation.

SR-IOV More recently, the limitation of HV VMs with regard to I/O (*i.e.*, lowered performance due to the need for paravirtualized I/O devices), has been supplanted by newer hardware with the *single root input/output virtualization* (SR-IOV) specification [38].

SR-IOV technology allows for PCI Express devices to be logically isolated for reasons of performance and maintainability. Under SR-IOV, a single physical device can be shared by exposing multiple, independent *virtual functions*. These virtual functions give the illusion of having multiple hardware devices, although they are associated with a single physical machine. The multiplexing of I/O streams is performed directly by the hardware and no I/O scheduling is required in software.

This is advantageous for virtualization as it allows different VMs to share a single I/O hardware device (*e.g.*, a network adaptor). In particular, combining SR-IOV with hardware-assisted virtualization eliminates the need for paravirtualized I/O entirely. However, as SR-IOV requires specialized I/O hardware, most hypervisors like Xen are typically configured for paravirtualized I/O out-of-the-box, with SR-IOV requiring additional configuration.

One potential downside to SR-IOV is that it takes away software control of I/O scheduling and leaves the multiplexing of multiple streams of I/O from different virtual machines to the physical device itself, which typically uses a simple round-robin scheme by default. This makes it difficult to enforce more complex I/O policies such as prioritizing I/O traffic from specific VMs. However, in such cases where more complex policies are required, paravirtualized I/O is still a viable solution.

2.1.3 Intel VT-x and AMD-V

Newer Intel processors provide virtualization support via the VT-x extensions. The VT-x extensions add an Input/Output Memory Management Unit (IOMMU) that enables virtualized systems safe, direct access to the memory of I/O devices. It also enables *direct memory access* (DMA) and interrupts for devices to be directly mapped into VMs in an efficient manner (*e.g.*, for Ethernet and graphics devices). Finally, *extended page tables* enable direct translation from guest virtual addresses to physical addresses without the need for shadow page tables in software.

AMD processors support virtualization through a similar set of hardware extensions called AMD-V. The AMD-V technology introduces similar technologies as VT-x. For example, *rapid virtualization indexing* (RVI) assists with virtual-to-physical page translations, similar to VT-x's extended page tables).

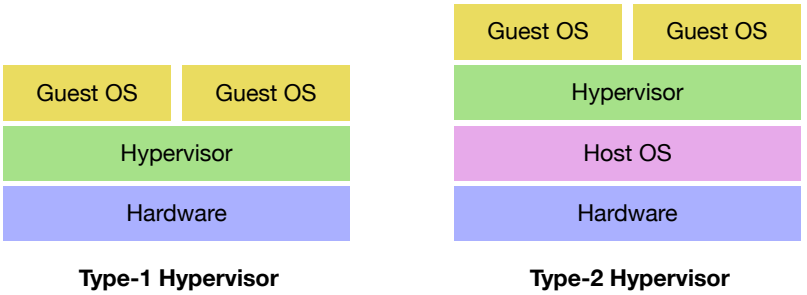


FIGURE 2.1: The two types of hypervisors

VMs using hardware-level features for virtualization such as those provided by Intel VT-x and AMD-V have come to be commonly known as *hardware virtual machines* (HVM) and have seen widespread adoption in cloud infrastructure. Therefore, throughout this thesis, when we refer to VMs, we specifically refer to HVMs unless explicitly specified.

2.1.4 Types of Hypervisors

A common distinction is made in prior literature to two types of hypervisors: type 1 and type 2.

Type-1 hypervisors run on the physical hardware, interfacing with it directly. The hypervisor, which is installed on the hardware, typically boots a supervisory host OS at boot within a VM, except it provides it with privileged access to the underlying hardware. Examples of this type of architecture include Xen and VMWare ESX Server. In Xen terminology, the hypervisor hosts a privileged supervisory VM called Dom0, and a set of guest VMs called DomU's. DomU's are provided with access only to the resources allocated to it via Dom0.

Type-2 hypervisors run as a regular software program on top of a regular OS and typically provide software virtualization (although some like QEMU can make use of hardware extensions by interfacing with a type-1 hypervisor running beneath the host OS). The advantage of type-2 hypervisors is that they can be ported to any OS just like any other piece of user software and does not require modification of the host OS. VMWare Server, Oracle's Virtualbox, and QEMU are examples of type-2 hypervisors. The downside of this approach is lower performance due to emulation overheads and the presence of an OS that intermediates between the hypervisor and the hardware.

A comparison of the two types of hypervisors are shown in Figure 2.1. In this thesis, we focus on Xen, which is a type-1 hypervisor. However, we emphasize that the techniques presented here are in no way limited to them and can be employed in type-2

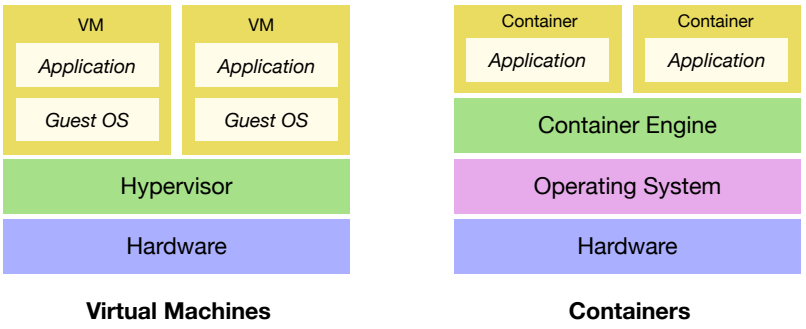


FIGURE 2.2: A comparison between virtual machines and containers

hypervisors as well.

2.1.5 VMs vs. Containers

Containers are a lightweight alternative to VMs that virtualize the system at the OS API level rather than that of the instruction set architecture as is the case with VMs. While both essentially provide isolation between individual components, containers have seen an increase in adoption in commercial cloud environments today due to their smaller resource consumption and faster boot times.

Figure 2.2 shows the key differences between VMs and containers. As can be seen in the figure, containers are more lightweight, in terms of resource usage, compared to VMs. This is due to VMs having an entire guest OS, whereas a single guest OS is shared across all containers in the system. Containers are typically implemented by using kernel-level support for isolating resources among processes via namespaces, where each process only sees its own environment and does not have access to those of other processes running in the system.

Owing to their lower runtime overhead, containers work well for running a large number of applications on a specific OS. In comparison, the number of VMs that can be supported on the same hardware is significantly lower due to the higher resource usage from having a distinct guest OS in each VM.

The disadvantage of container-level isolation is that sharing a kernel introduces a much larger attack surface that is more prone to exploits from malicious applications that are co-located on the same machine. Therefore, for applications that require stronger security and isolation across tenants, VMs are desirable due to their stronger isolation.

In this thesis, we focus exclusively on VMs. As we show later in section 3.2, Tableau's design results in increased creation and teardown overheads, making it unsuitable for

scheduler container-based workloads. However, we present multiple techniques in section 6.1 that can be employed by Tableau to mitigate this overhead. Further, any improvements in the efficiency of the hypervisor control plane can further lower these overheads. For example, LightVM [80] is a Xen-based virtualization solution that features a redesign of the Xen's control plane, called Tinyx, which allows custom-built VMs to boot up in a few milliseconds. Such improvements can be immediately combined with Tableau to adapt it for containers. In fact, LightVM shows that the primary benefits of VMs over containers (*i.e.*, strong isolation) can be achieved with low overhead. For example, VMs under LightVM can be booted two orders of magnitude faster than Docker (a popular containerization platform [87]) and comparable to the overhead of a `fork()` and `exec` on Linux.

2.1.6 VMs and Unikernels

As we have seen above, VMs have the downside that they require more resources owing to the fact that each VM has a complete guest OS embedded inside it. While containers solve this by virtualizing at the OS API level and sharing a guest OS among all containers, it comes at the cost of increased attack surface since any container that compromises the kernel gains privileged access to the entire machine. This can include attacks that target bugs in the exposed kernel API or simply just denial-of-service attacks that work by exhausting shared system resources (*e.g.*, memory, file descriptors).

One approach to avoid this is to run multiple containers from a single client inside a client-specific VM.

An alternative approach is to use *unikernels*, which are a recent attempt to combine the advantages of both VMs and containers. That is, to provide fully-isolated virtual machines with a smaller attack surface, while also ensuring low resource usage.

Unikernels such as MirageOS [76] achieve this by building application-specific virtual machines by bundling the application and its dependencies into a lightweight VM image. By doing so, the attack surface is reduced as there is no guest OS but merely a thin library OS that mediates access between the applications, its dependencies, and the hardware. Further, the absence of a full guest OS in each VM means that VMs can be booted significantly faster and incur lower resource usage.

Unikernels are still an active area of research and, at the time of writing this thesis, no major cloud provider provides a commercial offering. However, while this thesis focuses on traditional VMs (*i.e.*, with a full guest OS per VM), most of the techniques presented can be adapted to unikernel-based VMs as well. In some cases where it is

not possible to directly adapt Tableau to support unikernels, we explicitly point it out and present alternatives.

2.2 The Xen Hypervisor

Xen is a commercial, open-source type-1 hypervisor that we use in this thesis to implement the design of Tableau. The three core components of a Xen setup are the hypervisor, the supervisory VM, and guest VMs.

The hypervisor runs on the hardware, and arbitrates physical resources among multiple VMs running on it. While Xen was originally designed as a paravirtualized hypervisor, it has evolved over the years to make use of hardware support for virtualization and supports HVM guests. However, as we describe below, I/O under Xen is still paravirtualized by default as it avoids the need for specialized hardware, and has the added advantage that one does not have to implement a large number of device drivers, and can piggyback on the range of drivers available in dom0.

Dom0. At boot, Xen spawns a supervisory VM (called *domains* in Xen terminology), called *Domain-0* (Dom0 for short), that receives privileged access to hardware via the hypervisor. This enables it to issue *hypercalls*, a mechanism for interacting with the hypervisor analogous to system calls in an OS, in order to **(i)** manage and configure new and existing VMs as well as **(ii)** configure the system (*e.g.*, setting resource limits for individual VMs and assigning PCI devices to other VMs). The VMs created via Dom0 are called guest VMs or *DomU's* in Xen terminology.

In practice, Dom0 runs Linux, although Dom0 implementations also exist for other OSes such as OpenSolaris [85] and NetBSD [19]. For this thesis, we assume the common-case of a Linux-based Dom0.

Dom0 communicates with the hypervisor via a *hypercall* interface that parallels how userspace processes communicate with an OS kernel using a system call interface.

As described above Dom0 acts as the control plane of a Xen-virtualized machine. However, Dom0 also has another role in Xen: as the hypervisor itself does not have any device drivers, Dom0 handles I/O for all VMs in the system by default. This is implemented via high-performance virtual device drivers in guest OSes (called frontend drivers) that allow data to be passed between guests and Dom0 using shared memory, with the guest running the backend driver. The backend driver is responsible for translating I/O requests into a device-native request and for forwarding it to the actual device driver.

As a consequence of this architecture, Dom0 has the responsibility of multiplexing the I/O requests from multiple VMs onto a single hardware device. Typically, this means that guest I/O can be configured using Linux's standard interfaces for configuring I/O.

The advantage of this design is that the hypervisor itself can stay fairly lean and does not need to provide device drivers for all the hardware available, and can instead piggyback on Linux's mature driver-support codebase. Further, it also negates the need for more expensive hardware that supports I/O multiplexing via SR-IOV.

DomU's. While Xen runs a single Dom0 for the entire system, it is capable of hosting multiple unprivileged guest VMs, also called DomU's. DomU's are restricted in what actions they can perform. Typically this means that DomU's do not have access to any hypercalls that access hardware (only Dom0 can do that) unless explicitly permitted to do so (*e.g.*, by Dom0 configuring device passthrough for a given VM). As a result, in the typical case, where device passthrough is not configured, the guest OS running in DomU's does not run drivers for the hardware device, but simply the frontend component for Xen's special I/O devices that allow for communciating I/O requests to Dom0.

One disadvantage of this approach is that Dom0 is part of the entire *trusted computing base* of the system, in that, compromising Dom0 gives an attacker unfettered access to the entire system, including all VMs hosted on it. It should be noted that Xen provides an alternative approach for hosting device drivers in separate VMs instead of in Dom0 via a library-based operating system called *minios*. This allows for reducing the surface area for attacks by using a dedicated device-driver VM, which if compromised doesn't give access to Dom0. Therefore, the security of Dom0 is not an insurmountable architectural security issue in Xen, but a practical design choice. In this thesis we assume a Xen setup with Dom0-based device drivers.

2.3 VM Scheduling in Xen

The role of a VM scheduler is analogous to the role of a process scheduler in an OS. Similar to how the latter allocates CPU time for processes in the OS, the former allocates CPU time to *virtual CPUs* (henceforth *vCPUs*). From a scheduler's perspective, a VM is essentially a container for multiple vCPUs. Therefore, a single-core VM contains a single vCPU, while a multi-core VM consists of multiple vCPUs. Regardless, the scheduler need only be aware of vCPUs and their requirements.

A VM scheduler must satisfy two requirements. First, it must ensure that all vCPUs in the system receive *the right amount* of CPU time. We refer to this as the *utilization* of a

vCPU, and is typically configured by system administrators. Second, a VM scheduler must ensure that vCPUs receive their utilization *in a timely manner*. That is, the maximum delay between consecutive slots when a vCPU is scheduled should be bounded in some way. To illustrate this, consider two scenarios, each consisting of two vCPUs with a utilization of 50%. In the first scenario the VM scheduler alternates between the two VMs, running each one continuously for 10ms. In the second scenario, the VM scheduler does the same thing but alternates between them every hour. In both scenarios, while the long-term utilization is identical (*i.e.*, 50%), the *scheduling delay* (*i.e.*, the delay introduced by the scheduler) is significantly different, and leads to different performance characteristics. For example, while the former scenario would be suited for a latency-sensitive VoIP workload, the latter would not.

Xen comes bundled with multiple schedulers, namely Credit, Credit2, and RTDS. In this thesis we consider these three owing to their being the most mature. This choice of schedulers that Xen provides is thanks to a built-in scheduler framework that allows for easily implementing new schedulers. In this section, we will look at the scheduler framework in Xen in and provide a brief description of each of the above three schedulers.

2.3.1 The Xen Scheduler Framework

Xen comes bundled with an extensible scheduler interface, which provides generic glue code that is common to all scheduler implementations and scheduling algorithms. Using the framework, a new scheduler may be implemented in a modular fashion, with interaction between the generic parts of the scheduler, and specific code pertaining to a given scheduler implementation occurring via a well-defined scheduler interface.

The scheduler interface is implemented in the form of an abstract interface comprised of a set of functions pointers, which can be overridden to inject custom scheduler-specific behavior into key parts of the VM creation, configuration, teardown, and scheduling control flows.

In order to implement a new scheduler, one must simply instantiate a structure with pointers to functions containing scheduler-specific logic. This structure can be added to the list of available schedulers, allowing for the new scheduler to be selected at boot time. Below, we give a high-level overview of the callbacks that were relevant to Tableau's implementation.

Initialization and teardown. The `init` and `deinit` callbacks are invoked to provide schedulers with an opportunity to initialize and teardown global structures pertaining

to the scheduler itself, and must be overridden by each new scheduler implementation.

Following this, each scheduler must instantiate structures to deal with **(i)** each physical core in the system, **(ii)** each VM in the system, and **(iii)** each vCPU in the system (recall that in Xen, VMs are simply containers for one or more vCPUs).

To deal with physical cores, the `alloc_pdata()` and `free_pdata()` callbacks are invoked to allow schedulers to allocate and deallocate per-CPU structures, while the `init_pdata()` and `deinit_pdata()` callbacks are invoked to allow for their initialization and teardown.

A similar approach is employed for vCPUs, where for each vCPU in the system, the `alloc_vdata()` callback is invoked upon a new vCPU being created, and the `free_vdata()` callback is invoked before it is removed from the system. Similar to the callbacks for physical cores, the `insert_vcpu()` and `remove_vcpu()` are invoked to initialize vCPUs after allocation, and teardown vCPUs before deallocation, respectively.

Finally, individual VMs (called domains in Xen) are created and torn down via the `alloc_domdata()` and `free_domdata()` callbacks.

Dealing with blocking and wakeups due to I/O. Schedulers must also implement logic for dealing with blocking and waking up of vCPUs when I/O operations occur or complete, respectively. The Xen scheduler framework invokes the `sleep()` callback when a vCPU blocks and the `wake()` callback when it is woken up. It should be noted that, due to how Intel processors route interrupts, wakeup events for a particular vCPU may occur on any core in the system, and not necessarily on the same core where the VM originally blocked. This means mutual exclusion (using spinlocks) must be ensured when accessing and modifying vCPU state in the `wake()` callback. Further, the `wake()` callback can specify whether the scheduler needs to be invoked at the end, allowing for a new scheduling decision to be made following the unblocking of a vCPU.

Scheduling logic. Finally, a scheduler must implement the logic for dispatching vCPUs onto cores, for which the `do_schedule()` callback can be provided. The callback must be a function that returns a vCPU structure along with the time that it should be allowed to run for (including, if necessary, indefinitely). The scheduler framework then saves the state of the previous vCPU, and the new vCPU returned by `do_schedule()` is dispatched. If a finite amount of time was specified for the vCPU to run, a timer is programmed to invoke `do_schedule()` again after the provided time interval.

Finally, once the state of the currently-running vCPU as been saved, the `context_saved()` callback is invoked. This is where logic such as migrating vCPUs can be performed as doing so directly in the `do_schedule()` callback, before the state of the de-scheduled vCPU has been saved, may result in multiple cores accessing it simultaneously, leading to stack corruption.

We now provide a brief description of the three Xen schedulers considered in this thesis.

2.3.2 The Credit Scheduler

The Credit scheduler is a proportionate fair-share scheduler aimed at ensuring fairness and low latency for I/O applications. At the time of writing this thesis, the *Credit scheduler* is the default scheduler in Xen [5].

Each VM running under Credit is associated with a *weight* and *cap*. The former determines the proportionate share of CPU time that each vCPU belonging to the VM receives, while the latter determines the upper bound (*i.e.*, vCPUs are cut off from further CPU time once the cap is reached).

Under Credit, VM weights are relative, which means that the value is only relevant in proportion to other VMs in the system. For example, all VMs having a weight of 256 (the default) means every vCPU gets an equal share of CPU time, but is also true if all VMs have a weight of 512, or 1024. However, two VMs with a weight of 256 and 512, respectively, will result in the latter getting double the share of CPU time compared to the former. Note that for vCPUs, the maximum time possible is an entire core, and while we talk in terms of fractions of a core, we are are only referring to the *total* CPU time provided to a VM under Credit. In reality, this CPU time may be distributed across multiple physical CPUs (unless the VM has been pinned to a specific core).

VM caps under Credit are absolute and are representative of the maximum proportion of a physical CPU that it can use. For example, a cap of 50 means that all vCPUs for that VM will each be limited to at most 50% of a single core. Specifying a cap of zero (the default) implies there is no upper limit on CPU share for a given VM, and each of its vCPUs can use up to 100% of the physical core if available.

Under the default settings (no cap in place), Credit is work conserving; that is, whenever there are free cycles on a particular core, runnable vCPUs will be dispatched to execute on it. Setting a cap on a VM, however, may result in non-work-conserving behavior. Specifically, when all VMs in the system are capped, and the sum of these caps is less than the total CPU share available, there is guaranteed to be idle time. However, note that having even just a single uncapped, runnable vCPU per core is sufficient to "soak up" any idle cycles, making the system work-conserving again.

The Credit scheduling algorithm. Under Credit each vCPU is provided with *credits* proportional to its weight as discussed above. At any given time, a vCPU can be in one of two states, OVER or UNDER, representing whether a vCPU has credits available or has used up all its credits, respectively. Credit schedules vCPUs in the UNDER state in a round-robin fashion, and if none are available on a given core, it attempts to pull UNDER vCPUs from other cores in the system. Once a vCPU exhausts its credits, it is put into the OVER state.

The Credit scheduler schedules vCPUs in 30ms quanta by default. That is, vCPU's are dispatched to run for 30ms at a time, and once the quantum *expires*, a new vCPU is chosen to run. It also periodically (by default every 10ms) performs an accounting cycle where vCPU's priorities are updated based on its CPU usage pattern (see below).

Once all runnable vCPUs have exhausted their credits, and only OVER vCPUs remain, Credit replenishes all of their credits based on their proportional weight, and moves them into the UNDER state so they can be dispatched again. Note that only runnable vCPUs are replenished.

In the case of capped domains, there may be a situation where a vCPU has credits but has hit its cap, and therefore cannot be scheduled. In this case, when new credits are allocated, Credit accounts for this by taking the credits that would have gone to the capped vCPU, and distributing them to the other runnable ones. That is, it calculates the distribution of credits assuming the capped VM is not runnable.

Finally, to improve I/O latencies, Credit introduces a third priority level called BOOST. A vCPUs is set to the BOOST priority as soon as it is woken up (*i.e.*, as a result of an I/O completion), and any boosted vCPUs are given a higher priority than regular UNDER vCPUs. Credit's periodic accounting tick is used to remove the BOOST state. Essentially, if a boosted VM is detected to be executing for more than a single tick, it is marked as a CPU-bound workload, otherwise its BOOST state is maintained. In effect, the periodic tick allows Credit to determine whether a VM is expressing characteristics of short I/O-related work or longer CPU-bound work.

Finally, Credit also attempts to balance vCPUs across cores via a *pull* mechanism. This means that a core that idles will attempt to acquire and dispatch runnable vCPUs on other busy cores.

2.3.3 RTDS Scheduler

The Real-Time Deferrable Server (RTDS) scheduler is a real-time scheduler in Xen that is aimed at workloads that require predictable latency characteristics. The RTDS scheduler is derived from the RT-Xen project [120, 122], which is an extension of Xen with

support for an assortment of real-time scheduling schemes, created with the goal of supporting real-time guarantees in virtualized clouds.

The RTDS scheduling algorithm. Under RTDS, VMs are configured using a period (P) and budget (B), and each vCPU in the system is guaranteed up to B units of CPU time within a period of time P. Note that the budget need not be contiguous and is available to the VM at any time during its period. At the end of each period, any remaining budget is discarded and then fully replenished to B units. Therefore, the ratio of B and P specifies the utilization of the VM. Each VM also has an implicit deadline (*i.e.*, at any given time, the deadline is equal to the end of the current period's time interval).

VMs under RTDS are therefore eligible for dispatch only when they have a non-zero budget, and all eligible VMs are maintained in a global ready queue. RTDS schedules VMs using a global earliest-deadline-first (G-EDF) scheduling scheme, where VMs with earlier deadlines are prioritized over those with later ones, and at any given time, the m highest-priority VMs are scheduled on the m processors in the system.

While a VM is running on a CPU, its budget is depleted continuously until it runs out of budget or is replenished at the end of the current period's time interval.

As we will see later in Chapter 5, Tableau models VMs in a similar way as RTDS, but takes a very different approach to scheduling VMs. RTDS focuses primarily on predictability while Tableau additionally optimizes for high throughput by lowering the runtime overheads of its implementation. In contrast, for example, RTDS' implementation uses a global lock to ensure mutually exclusive access to its global ready queue, which results in higher runtime overheads due to increased lock contention, thereby lowering throughput.

2.3.3.1 The Credit2 Scheduler

The Credit2 scheduler is the successor of Xen's Credit scheduler and was was designed with particular focus on improving support for mixed workloads (*i.e.*, batch and latency sensitive) and improving latency of virtualized applications.

The documentation and technical details of the Credit2 scheduler are sparse, and its implementation in the version of Xen that we based the implementation of Tableau on (Xen 4.9) did not support many of the configuration options that the original Credit scheduler did. However, we still evaluate it extensively in Chapter 6 and show that, while it provides improvement in latency metrics for certain workloads compared to Credit, it achieves lower throughput.

One primary change compared to Credit is the removal of BOOST states for VMs. VMs are only scheduled based on their position in the ready queue, which itself depends on remaining budget.

As of writing this thesis, the Credit2 scheduler was still under development, with the older, more stable Credit scheduler being the default [5].

2.4 VM Scheduling in Other Hypervisors

Scheduling in KVM. Kernel-based Virtual Machine (KVM) is a virtualization solution for Linux running on x86 and x86_64 hardware. It is provided as a loadable kernel module that allows a regular Linux kernel to function as a hypervisor. As a result of KVM's tight integration with the Linux kernel, VMs are scheduled like any other process using Linux's in-built scheduling framework.

Linux's CFS scheduler. The default Linux scheduler is the *Completely Fair Scheduler* (CFS), which is the default scheduler used by KVM to schedule VMs. While CFS has many improvements built on top of its scheduling (*e.g.*, group- and user-fair scheduling, as well as a modular scheduler framework [116]), we focus on the core scheduling approach below.

CFS is a proportionate fair-share scheduler, similar to Xen's Credit scheduler. Under CFS, vCPUs are dispatched based on a "virtual runtime", which also determines their priority. Virtual runtime simply tracks the amount of time a given vCPU has spent executing on the processor. The longer a particular vCPU has executed on a processor, the lower its priority, allowing all vCPU to execute fairly over time. For improving the performance of interactive tasks, CFS modifies the virtual runtime value of vCPUs that block. Upon unblocking, the waking vCPU is inserted back into the ready queue (implemented as a red-black tree) with its virtual runtime adjusted to prioritize it higher [115].

At an implementation level, CFS uses per-processor ready queues similar to a partitioned real-time scheduler [34], and each vCPU belongs to exactly one ready queue at any given time. However, CFS approximates a global scheduling approach [34], where a single global ready queue holds all vCPUs in the system and each processor dispatches vCPUs from it. It does this approximation using a push-pull mechanism where vCPUs are migrated at runtime to ensure that, for a system with m processors, the m highest-priority vCPUs are always executing. We describe both the push and pull mechanism below, and how it is used to approximate a global scheduler.

CFS push operation. A push operation is performed by CFS when **(i)** a currently-suspended, higher-priority vCPU resumes execution, or **(ii)** when the currently-executing vCPU is preempted by a higher-priority one. The scheduler iterates over the ready queues of all processors in the system and attempts to "push" the currently-running task onto their local ready queue. The core to migrate to is chosen based on whether it is executing a vCPU with a lower priority that that of the vCPU being migrated.

CFS pull operation. A pull operation is performed whenever a core schedules a lower-priority vCPU, such as when a previously-running higher-priority vCPU blocks or is torn down. In this case, the scheduler again scans the ready queues of remote processors and tries to find higher-priority vCPUs to schedule instead. If any such vCPUs are found, the highest-priority one is migrated. The vCPU is chosen such that it has a priority higher than all vCPUs in the local ready queue.

Together, the push and pull operations enable CFS to use distributed ready queues to approximate the invariant of a global scheduler, which suffers from increased average-case overheads due to lock contention on the global ready queue (as is the case for RTDS). However, it has been shown that while CFS performs well in the average case, in a system under load, it induces high worst-case latencies due to requiring more complex locking of local ready queues [23].

Scheduling in VMware ESXi. VMware ESXi is a commercial type-1 hypervisor that is used in commercial cloud environments [3].

Under ESXi each virtual machine, or *world*, comprises one or more vCPUs. VM scheduling in ESXi uses a proportionate fair-share scheduling algorithm like Xen, which allocates CPU time to worlds based on a combination of three user-specified parameters: *shares*, *reservations*, and *limits* [107].

Shares are similar in nature to Xen's VM weights and determine the CPU time allocated to a VM proportional to the total number of shares in the system. For example, if VM_1 has twice as many shares as VM_2, it would receive twice as much CPU time as VM_2. Reservations specify a guaranteed lower bound on CPU time for a VM, while limits specify an upper bound in CPU time that can be allocated to a VM. It should be noted that under ESXi, VMs can be assigned to a group in a configurable hierarchy of groups, similar to CFS, and configuration parameters can be specified at the group level directly. However, for the sake of clarity, we limit the description below to the simpler case of VM-level configuration below.

The ESXi scheduler uses a quantum-based scheduling approach, where the quantum is a globally configurable value, similar to the global quantum in Xen's Credit scheduler. The VM scheduler is invoked whenever the currently running world blocks, or its time

quantum expires. When invoked, the ESXi scheduler schedules the next ready world from a local ready queue. If no world is found locally, a remote ready queues are searched, and if even that fails, the idle world is picked.

Worlds are charged budget when they execute, and are prioritized in the ready queue in decreasing order of remaining budget. As a result, generally, a world that is frequently blocked due to I/O gets scheduled more promptly compared to a world that is CPU bound, as the latter uses its budget more compared to the former within any given time interval. In this regard the ESXi scheduler has a similar design to Xen's Credit2 scheduler, and does not have any special boosted priority for I/O intensive VMs,

2.5 Real-Time Scheduling

Next, we review necessary background in the area of real-time systems pertaining to the work in this thesis.

2.5.1 Periodic Task Model

Periodic tasks are a classic real-time task model [72] where a task τ_i is characterized by two parameters (C_i, T_i): its worst-case execution time C_i, and its period T_i. A periodic task τ_i is assumed to *release* a *job* every T_i time units, with each job taking *at most* C_i time units anywhere during the current period's interval. The only associated correctness criterion is that each job released by a periodic task must receive (up to) C time units of processor service during each scheduling interval $[0, T)$, $[T, 2T)$, $[2T, 3T)$, *etc.*. That is, periodic tasks are assumed to have an *implicit deadline* where the deadline D_i for a task τ_i is equal to its period T_i.

2.5.2 Schedulability, Feasability, and Optimality

A given task set is considered to be *schedulable* if the temporal constraints of all tasks comprising it will always be satisfied (*i.e.*, no deadlines are missed). A task set is *feasible* if and only if there exists a scheduling algorithm such that the task set is schedulable under it. Finally, a scheduler is considered *optimal* if it can successfully schedule all feasible task sets. That is, a scheduler S is optimal if and only if every feasible task set is schedulable under S.

2.5.3 Partitioned Earliest-Deadline-First Scheduling

Earliest-deadline-first scheduling. *Earliest-deadline-first* (EDF) is a dynamic-priority real-time scheduling algorithm. Under *earliest-deadline-first* (EDF) scheduling, jobs are dispatched in order of increasing deadlines (*i.e.*, the more urgent a task is, the earlier it is dispatched), with no prior priority assignment required. Liu and Layland, showed that EDF is optimal on preemptive uniprocessors. When scheduling periodic tasks with implicit deadline (*i.e.*, tasks that have deadlines equal to their periods), EDF has a utilization bound of 100%. That is, EDF guarantees that all tasks meet their deadlines provided the total CPU utilization is less than or equal to 100%.

Partitioned EDF. There are three types of multiprocessor scheduling: global, partitioned [37], and clustered. Under global scheduling, all processors dispatch jobs of tasks from a single ready queue, and jobs may migrate freely among processors. Under partitioned scheduling, tasks are statically assigned to processors during an offline, partitioning phase, with each processor scheduling all jobs of tasks assigned to it using a uniprocessor scheduling policy. Finally, clustered scheduling is similar to partitioning, except the size of each partition is greater than one (*i.e.*, not just a single processor). Alternatively, partitioned scheduling can be considered a special case of clustered scheduling where the cluster size is equal to one. *Partitioned EDF* in particular (P-EDF) is a partitioned scheduling policy where each processor schedules all tasks assigned to it using an EDF policy.

Partitioning heuristics. While partitioned multiprocessor scheduling has the advantage that each processor can be analyzed as a uniprocessor, it comes at a cost: in order to obtain a partitioned system, the task set needs to be first partitioned. That is, tasks must first be assigned to individual processors such that neither of them is overloaded. Unfortunately, solving this task-assignment problem requires solving a bin-packing-like problem, which is a classic intractable problem that is NP-hard in the strong sense [37, 44]. However, as a result of this, we can use existing bin-packing heuristics to partition task sets. Prior work on bin-packing heuristics is extensive and we do not attempt to summarize it here as it is beyond the scope of this thesis. Instead, we refer the interested reader to a survey of the same [27].

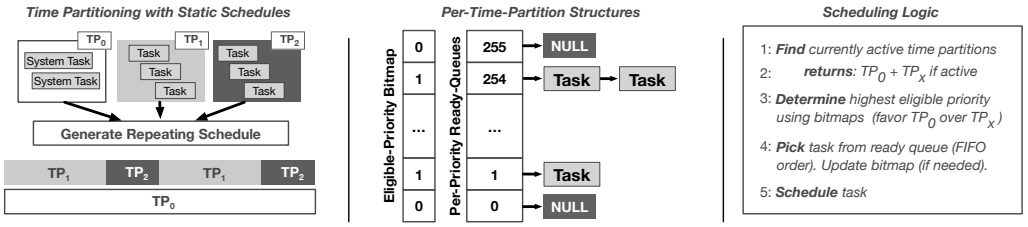


FIGURE 2.3: An overview of the PikeOS scheduler. The system comprises multiple application time partitions (TP_X), for which a static schedule is generated, and the system time partition (TP_0), which is always eligible (left). Each time partition consists of 256 ready queues and an associated bitmap tracking which levels have eligible tasks (middle). At runtime, the first task from the highest-priority, non-empty ready queue, in either TP_0 or the currently active TP_X, is dispatched (right).

2.5.4 Space and Time Partitioning with ARINC 653

Table-driven scheduling is not a new technique; it was originally proposed by Kopetz and Bauer in their Time-Triggered Architecture [65]. One of the more prominent applications of table-driven scheduling is the ARINC 653 software specification for space and time partitioning mandated in safety-critical avionics RTOSes. Under ARINC 653, tasks are partitioned into cores and a repeating table is generated that schedules each task periodically. This simple design facilitates the predictability mandated for critical avionics software and Tableau applies a similar approach to VM scheduling to improve predictability. [1]

To give a more concrete example, we describe a real-world ARINC 653-compatible scheduler from PikeOS, a certified microkernel designed for safety-critical applications.

2.5.4.1 Example: Scheduling in PikeOS

PikeOS is a separation microkernel for multi-core, hard-real-time systems, and can be used as both a real-time OS (*i.e.*, hosting native applications) as well as a type-1 hypervisor (*i.e.*, hosting complete operating systems). PikeOS provides various *personalities*, or different OS interfaces (*e.g.*, ARINC 653, Linux, POSIX, AUTOSAR, *etc.*), for the development of applications in different domains.

PikeOS is widely used in industry due to its certifiable nature: PikeOS has been certified to safety standards such as DO-178B (avionics), IEC 61508 (electrical/electronic/programmable electronic safety-related systems), and EN 50128 (railways), making it a "battle-hardened," top-tier choice for mixed-criticality applications with components at different safety and security levels that need to be isolated via resource partitioning.

[1] In fact, Xen contains an ARINC 653 scheduler implementation, although it is limited to uniprocessors and has not been applied, to the best of our knowledge, in the context described in this thesis.

Scheduling in PikeOS. The PikeOS scheduler is based on time partitioning (as defined by the APEX specification in the ARINC 653 standard reference). Conceptually, time partitions are encapsulating containers for a set of threads, where threads in different time partitions are scheduled in mutually exclusive time windows. In PikeOS, application tasks are assigned to *application time partitions*, which are activated periodically as specified by a repeating static schedule. When a time partition is activated, any tasks of the previously active time partition are forcibly preempted; frequent time-partition switches thus incur significant runtime overheads. As application time partitions are strictly separated from another, tasks in different partitions can be certified to different levels of assurance (*i.e.*, each in accordance to its own criticality).

Owing to PikeOS' microkernel design, all OS functionality (*e.g.*, device drivers, file systems, *etc.*) is implemented as service daemons (or tasks) scheduled in time partitions similarly to standard applications. To ensure that essential system services are always available, PikeOS assigns these tasks to a special *time partition zero* that is always eligible to run. Thus, in PikeOS, there may be up to two active time partitions in the system at any given time: time partition zero (TP_0) and, depending on the static schedule, an application time partition (TP_X). Note that threads in TP_0 are always certified to the highest assurance level since their functional and non-functional correctness is essential to the correct operation of the entire system.

For every time partition in the system, PikeOS maintains a ready queue for each of 256 supported priority levels, and each ready queue is simply a FIFO-ordered list of tasks eligible for scheduling at that priority level. When making a scheduling decision, the scheduler finds the highest-priority, non-empty ready queue, and picks the first task from it. It does this taking into account tasks from both TP_0 and the currently active TP_X, if any, with TP_0 tasks taking precedence over those in the currently active TP_X at any given priority level. Consequently, while the worst-case latency incurred by tasks in TP_X depends on the static schedule, TP_0 tasks are always schedulable and thus incur lower latency (affected only by other higher-priority tasks in the system).

For each time partition in the system, the PikeOS scheduler maintains a *priority bitmap* to track non-empty priority levels so that it can efficiently determine the highest priority that is currently eligible when making a scheduling decision. That is, for each time partition and priority level, a bit is set if and only if runnable threads exist in the corresponding time partition at the given priority level.

Figure 2.3 summarizes the PikeOS scheduling architecture. On a multiprocessor, it is instantiated on each core (partitioned scheduling).

2.6 Performance Monitoring on Intel Platforms

Modern processers come equipped with a Performance Monitoring Unit that enables collection of low-level processor events during its execution. Modern Intel processors support two categories of performance monitoring events; the first are a large set of model-specific events that vary from one processor model to the next, while the second are a smaller set of events that are consistent across processor models [52]. The PMU can be programmed to increment a counter when a specific event occurs as well as to raise an interrupt when a counter exceeds a user-specified value. The latter is useful for recording the instruction pointer, allowing for profiling a program using statistical sampling.

Various high-level libraries are available for using performance counters to measure the performance of native applications running on top of an OS such as Linux. These include, for example, Intel's VTune [98] profiler that allows for analyzing application performance using performance counters, the PAPI standard, which specifies a standard *application programming interface* (API) for accessing hardware performance counters [88], the oProfile sampling-based profiler for Linux [28], Intel's Performance Counter Monitor tool [51, 113], and Linux's perf tool [35].

Various approaches also exist for measuring the performance of virtualized applications running inside VMs (*e.g.*, XenoProf [86], Perfctr-Xen [90]). Xen also provides a virtualized PMU driver, which uses a save-and-restore mechanism for PMU registers enacted upon vCPU context switches, as well as a trap-and-emulate mechanism for PMU configuration registers. This allows for using standard tools, such as the `perf` tool [39], within guest OSes to access performance counter data.

The limitation of these approaches is that they do not provide per-VM counter measurements, but rather support either system-wide measurement (as in the case of XenoProf) or measurement for virtualized applications running inside the guest OS running in a VM (such as in the case of Perfctr-Xen or Xen's virtualized PMU). To track per-VM performance, we use a simple implementation in Chapter 7, which involves directly programming the PMU registers. See section 7.3 for details on how we program the PMU and use it in our implementation of Tableau.

2.7 Prior Work

In this section we present prior work related to the work presented in this thesis.

Supporting soft- and hard-real-time applications. We first present relevant work that focuses on supporting hard- and soft-real-time applications within VMs, and makes the case for using techniques from the area of real-time systems. While we present a small subset of this work here, we refer the interested reader to a survey by García-Valls *et al*. [43] who present related work focused on supporting real-time applications in the cloud, and the various technical challenges involved.

Cucinotta *et al*. [31] present the challenges involved in ensuring timeliness guarantees for virtualized real-time applications, and show how simple real-time scheduling techniques, such as reservation-based scheduling, can be employed to provide stronger guarantees on response times. Cucinotta *et al*. [32] show how hierarchical scheduling mechanisms can be used to improve the predictability of virtualized applications. They present an improved schedulability test for hierarchical real-time systems, and show results demonstrating improved predictability of a set of KVM-based virtual machines. Masrur *et al*. [82] present a fixed-priority VM scheduler implementation and show how such a system can ensure timing constraints of an automotive-inspired real-time control loop. Crespo *et al*. [30] present XtratuM, which is a type-1 hypervisor designed to meet safety-critical real-time requirements. It does this via a space- and time- partitioning approach that provides strong temporal and spacial isolation for VMs (and supports the ARINC 653 standard). In a similar vein, Danish *et al*. [33] present the scheduling framework in Quest OS, an OS capable of scheduling vCPUs, which allows for safely co-locating batch and I/O workloads by employing real-time servers [71].

Xi *et al*. [120–122] presented RT-Xen, which implements a wide range of real-time scheduling algorithms: both global and partitioned schedulers are implemented, with support for both static and dynamic priorities, and with VMs modelled as either periodic or deferrable servers. The RTDS scheduler in Xen, which we evaluate in this thesis, is derived from the RT-Xen project. Other work has built on top of RT-Xen, such as RT-OpenStack, which integrates RT-Xen with the popular OpenStack cloud management system [100]. RT-OpenStack extends OpenStack with a real-time resource management interface, in order to support the co-location of real-time and non-real-time VMs on the same hardware. Finally, with respect to KVM, Checconi *et al*. [24], similar to the work on RT-Xen, extend the Linux scheduler to support real-time reservation-based servers, thus allowing KVM-based VMs to use real-time servers.

Co-locating batch and I/O workloads in clouds. Next we present selected work that looks at the challenging problem of how to co-locate CPU-intensive batch workloads alongside latency-sensitive I/O workloads in the cloud.

VSched [70] looks at how to co-host batch and interactive VMs on shared hardware, while ensuring both a predictable utilization for the batch VM, and low latency for the interactive one. While VSched does not attempt to optimize throughput, and implements its scheduler as a user-space process resulting in high runtime overheads, the paper makes a case for using periodic real-time scheduling in hypervisors to achieve more predictable performance.

Kim *et al*. [62] present a VM scheduler which detects I/O-bound VMs, correlates incoming events with them, and optimistically boosts their priority so they can handle them with low latency.

Lee *et al*. [66] present an enhanced version of Xen's Credit scheduler that allow VMs to specify their scheduling latency requirements using a *laxity* parameter. With this explicit latency goal available for each VM, their enhanced version of the Credit scheduler is able to better prioritize latency-sensitive VMs. The work focuses more on soft real-time workloads such as telephony applications, and shows how the default version of the Credit scheduler is unable to achieve the required performance for latency-sensitive workloads.

Govindan *et al*. [45] reinforce the point we make in this thesis that providing the right amount of CPU time is not enough; it must also be provided in a timely manner. Similar to our findings, it traces the root cause of performance degradation in cloud workloads to scheduler-induced delays. They propose a new scheduling approach that preferentially schedules I/O-bound VMs over CPU-intensive ones, resulting in improved I/O latencies (albeit at the cost of increased short-term unfairness in CPU allocation).

Xen schedulers. Xen's various schedulers have been studied extensively in prior literature, especially the Credit scheduler owing to its widespread use. Next, we briefly summarize relevant work in this area.

Cherkasova *et al*. [25] presented one of the early works that compared the performance of three Xen schedulers: the *Borrowed Virtual Time* (BVT) scheduler, *Simple EDF* (S-EDF) scheduler, and the Credit scheduler. While the BVT and S-EDF schedulers are not actively maintained, one of the interesting findings they present regarding Credit is that it shows high CPU allocation errors: deviations of up to 10% from the configured utilization were observed, even over a longer time interval of three minutes.

In particular, Xen's Credit scheduler has been studied extensively in prior literature. Ongaro *et al*. [92] studied the effect of different scheduler configurations on I/O performance. Multiple papers [64, 92, 112] have also studied how the co-location of different types of workloads results in performance degradation under Credit. As a result, various designs have been proposed in prior work to remedy the shortcomings of the

Credit scheduler. For example, there have been multiple attempts at improving the performance of I/O-bound VMs that are latency sensitive [45, 61].

Yu *et al*. [126] present some limitations of the Credit scheduler in Xen with regards to predictability. One of their findings presented shows that using different values for weights, but with the same proportions, results in variation in observed application latencies.

Abeni and Faggioli [5] present an empirical analysis of Xen and KVM scheduler-induced latencies. They conclude that, for the evaluated workload, KVM guests can achieve worst-case latencies around $100\mu s$, while in the case of Xen, larger latencies are observed. They also find that the source is not scheduler-induced delays, but rather the overhead from Xen's interrupt-forwarding mechanism. We note that this paper does not present a high-density scenario, which exacerbates the scheduling latencies; we show that under a high-density scenario, scheduler-induced delay can indeed have significant impact on I/O tail latencies (see Chapter 6).

Gupta *et al*. [48] study the performance isolation mechanisms in Xen, and conclude that they do not correctly account for resources consumed within the hypervisor when doing work on behalf of VMs (*e.g.*, I/O processing). They present the design and implementation of a monitoring system that accounts for this resource consumption. In modern clouds, the use of SR-IOV-enabled I/O hardware means there is significant less work performed within the hypervisor (or Dom0) to service I/O requests for individual VMs; rather, VMs directly access virtualized devices, with the hardware performing the multiplexing of requests across VMs.

Finally, ERTDS [118] is an extension of Xen's RTDS scheduler that allows real-time VMs to exceed their budget if available. The goal is similar to Tableau's *uncapped VMs*, where VMs are allowed to use available CPU time past their guaranteed allocation in order to improve average-case performance, as long as it does not interfere with the performance of other VMs (see section 3.2).

Performance variability and degradation in clouds. Performance variability in commercial public clouds has been extensively studied in prior literature. It is a well-known problem that is an inevitable consequence of hosting multiple tenants running a wide range of workloads on the same hardware. For example, Iosup *et al*. [53] analyzed production traces from Amazon Web Services and Google App Engine and found that a significant proportion of the workloads exhibited periodic load patterns on the order of days and weeks. Similar findings were made by Leitner and Cito [67] who conducted an extensive empirical evaluation of the performance variability of four cloud

providers, and concluded, among other things, that multi-tenancy has a large impact on throughput and predictability.

Nathuji *et al*. [89] empirically show that consolidating VMs onto a multicore system with a shared last-level cache causes performance degradation. In particular, they make the point that while strictly partitioning shared resources, as is the case with software-based cache management techniques such as page coloring [104, 129], can improve predictability, it comes at a cost. First, complexity of the system increases as the partitioning schemes must be incorporated, not just for caches, but other shared resources such as memory bandwidth [127] as well. Second, resource partitioning requires the design and implementation of effective partitioning techniques, and owing to the temporal variability exhibited in public clouds, would require a more dynamic approach where resources are re-allocated over time as workload characteristics change. Finally, hardware sharing techniques like HyperThreading [81] are explicitly designed to increase resource utilization through increase sharing, at the cost of system predictability. Such shared mechanisms must be disabled in order to support a partitioning-based scheme. Instead, they advocate for designing adaptive resource allocation mechanisms in hypervisors that are aware of *Quality-of-Service* (QoS) requirements of individual VMs and which transparently provision resources to achieve them. They present Q-Clouds, a control framework that allows VMs to specify QoS requirements, and optimizes resource allocation for each VM over time to mitigate performance degradation due to interference.

Tableau's unique semi-offline design enables it to perform changes to the scheduling tables without increasing the complexity of the hypervisor itself. This means that Tableau makes it easy to support re-partitioning at runtime to mitigate dynamic cache interference via a userspace daemon like we do in Chapter 7. Therefore, by designing the scheduler with simplicity and high-performance in mind, Tableau is able to avoid introducing significant system complexity when implementing a dynamic re-partitioning scheme.

Tail latency and mitigation techniques. *Tail latency refers to the latencies at the tail end of the observed latency distribution*, typically in interactive or user-facing services. In particular, it gets harder to control tail latencies as systems scale up in size and complexity. In such systems, high-latency spikes that were relative rare at smaller scales, become more prominent. We present some prior work that look at this problem and propose various mitigation techniques.

Dean and Barroso [36] outline common causes for high tail latencies in online services and present techniques to mitigate their effect on system performance. They argue for

designing *tail-tolerant services* that are able to provide end-to-end tail latency guarantees regardless of the performance of individual components comprising the system. This includes techniques such as monitoring and blacklisting machines that induce high latencies, and hedging requests by issuing the same request to multiple machines and using the result of the first request that completes. In comparison, Tableau addresses specific sources of latency variability, namely scheduler-induced delays, and tail-tolerant techniques as proposed by Dean and Barroso are complementary to its goals. That is, they can be combined with Tableau to further improve predictability.

Xu *et al*. [125] studied the tail latencies of network requests in multiple Amazon Web Services EC2 datacenters, and found that the root cause was the co-scheduling of CPU-bound and latency-sensitive VMs on the same cores. Based on this, they present Bobtail, a system that detects and avoids problematic VM placement. In comparison, Tableau is able to co-locate latency-sensitive and CPU-bound VMs on the same cores as the latency requirements of each VM is "baked in" to the scheduling tables directly, and strictly enforced by the VM scheduler.

Li *et al*. [69] study the tail latency characteristics of multiple workloads on Linux and identify various sources of latency in the system. These include interference from background processes, poor I/O scheduling, and CPU power-saving features. In Tableau, background VMs are scheduled by a lower-priority scheduling tier, and thus are dispatched only when no table-driven VMs are eligible to run, causing less interference. Further, our use of an SR-IOV-enabled virtualized network card means that I/O scheduling policy does not affect network tail latencies.

Closely related to our work on Tableau is SageShift [103], which has similar goals: improving application latencies while increasing resource utilization. SageShift does this through a strict admission-control component, which determines whether a new VM's *service-level objectives* (SLOs) can be met. Once admitted, the VM's SLO is guaranteed via a VM scheduler that dynamically adjusts, both the utilization and scheduling delay of the VM at runtime. While a comparison with SageShift would be interesting, we were unable to locate any publicly available source code for the project.

Interference detection and mitigation in virtualized environments. We now present some prior work that looks at the problem of detecting and mitigating performance degradation due to interference from co-located VMs sharing the same hardware.

In general, any shared hardware resources are a source of interference, and a lot of prior work has studied the problem in more detail. We do not attempt to exhaustively list each one here, but rather present some representative work. We refer the interested reader to a survey by Abel *et al*. [4] of the performance impact from interference in

both bandwidth-based shared resource (*e.g.*, memory buses), and storage-based shared resources (*e.g.*, caches).

DeepDive [91] is a system that identifies and mitigates performance degradation due to interference in virtualized environments. The challenge of mitigating interference, from the cloud provider's perspective is that they do not have visibility into client VMs, and therefore cannot easily determine when interference occurs. Rather, DeepDive runs VMs in isolation first to build a model of their performance metrics, and monitors various low-level hardware performance counters for each VM in the system at runtime. If a deviation from the expected performance trend is detected, it runs a clone of the VM in isolation and monitors its performance to confirm that it was indeed being interfered with. If the performance in isolation varies from the performance observed at runtime, the VM is migrated to a different machine.

DejaVu [109] has similar goals as DeepDive: to identify interference and mitigate it at runtime. It does this by profiling workloads in isolation to build a *workload signature* for each, and clusters multiple workloads based on their runtime characteristics. At runtime, DejaVu observes VM behavior and classifies its workload by computing its workload signature. Finally, it computes the resources that each VM requires in order to achieve its service-level objectives.

In contrast to both DeepDive and DejaVu, we take a simpler approach in Chapter 7 and attempt to detect interference using low-level hardware performance counters alone.

Javadi and Gandhi [55] presented DIAL, a load balancer that clients can use without assistance from the cloud provider. DIAL works by detecting the performance degradation due to local interference from co-located VMs, directly within client VMs themselves. Following this, it re-distributing application load to other VMs experiencing lower interference. The goals of DIAL are orthogonal to that of Tableau, and can be combined to achieve lower interference across client VMs.

Similar to DIAL, other techniques have been proposed in the literature for "user-centric" interference detection [22, 56, 78, 79], which involve detecting interference directly within client VMs. Such techniques have the advantage that they do not require specialized software to be deployed by providers, and can be set up by end users themselves. Our cache-interference detection approach described in Chapter 7 is not a user-centric detection mechanism, and is implemented via a combination of a lightweight performance counter monitor directly in the hypervisor, and a userspace daemon in Dom0. However, we note that such user-centric techniques can be incorporated by clients to additionally detect interference within guests running on Tableau.

Zhang *et al*. [130] proposed CPI^2, a technique for detecting and mitigating interference

in multi-tenant clusters using performance counter data. The metric they propose to for detecting interference is *cycles per instruction*. CPI is a measure of VM performance that looks at the average number of cycles it takes to retire an instruction. That is, the number of cycles a VM takes until an instruction is completed and its results are committed in the architectural state of the system. In general, increased cache interference results in an increase in CPI, as instructions take longer to retire due to longer memory fetches holding up their completion. CPI can be efficiently measured by calculating the ratio of the number of unhalted CPU cycles (*i.e.*, cycles where the VM was executing on the core) to the number of instructions retired, both of which can be monitored accurately in the Intel architecture using low-level performance counters. While the work on CPI^2 was aimed at container based workloads, the use of CPI in virtualized environments presents some challenges. In particular, as there is a guest OS between the kernel and the virtualized application, an increase in CPI cannot necessarily be attributed to external interference; it can also occur as a result of self-interference within the VM being measured. We detail this and other issues with CPI as a interference-detection metric in Chapter 7.

Wang *et al*. [110] proposed VMon, a system that monitors VMs using hardware performance counters and quantifies the interference between them. It does this using the LLC miss rates of each VM, and by analyzing how it correlate to the VMs performance degradation. It does this by placing the VM in a sandbox, monitoring its resource requirements, and building an interference prediction model, which is then used in production to detect interference. They found that different applications exhibit different LLC miss rate patterns under interference. Similar to VMon, we use LLC miss rates as a detection mechanisms in Chapter 7. However, we attempt to detect interference using the low-level metrics alone.

Intel Cache Allocation Technology. More recently, Intel introduced *Cache Allocation Technology* (CAT), which provides hardware support for partitioning the LLC, with various cache-allocation approaches having been proposed based on it [42, 123]. We do not use CAT in our approach as it requires specialized hardware, and partitioning the LLC comes with a trade-off in the form of lowered application performance.

Containers and unikernels. Recently, there has been a shift towards more lightweight, container-based isolation [12, 49, 84, 96, 101, 117, 119] and unikernels [63, 75], which enable lightweight, purpose-built "OS-less" VM images.

With regards to lightweight containers, they do not invalidate Tableau's design—the Tableau approach can be easily applied to schedule containers instead of vCPUs, provided the containers are sufficiently long-running. That is, for systems where the configuration of application images is relatively static, Tableau remains applicable. In particular, combined with container-orchestration tools like Kubernetes [21], Tableau may be used to declaratively specify performance requirements of containers running on a cluster. With regards to unikernels, as they are designed to be lightweight and application-specific, combining them with Tableau would provide significantly increased performance predictability.

However, we note that Tableau, as presented in this thesis, is not applicable for certain uses of containers and unikernels (*e.g.*, on-demand spawning of containers to service individual requests [77]), as such use cases break Tableau's assumption that VM (or container instance) creation and teardown are relatively infrequent events, which is not the case in such scenarios. However, we do not see such techniques replacing traditional virtualization in the foreseeable future. Further, we present various techniques for extending Tableau to support such use cases (see section 6.1), including caching tables, or pre-generating fixed-sized slots in tables.

End-to-end predictability. From an end-to-end perspective, VM scheduling is not the only source of unpredictability and tail latencies in data centers (see Chapter 7). For instance, much prior work has dealt with network performance isolation within datacenters [7, 47, 54, 57, 94, 114]. An example of a system that integrates multiple techniques into a complete system is Heracles [73], which shows how to jointly consider multiple aspects (scheduling, memory isolation, and network isolation) using Linux's heuristic-driven CFS scheduler. The contribution of Tableau is to specifically improve the design of the VM scheduler, and can be combined with techniques that target other sources of unpredictability.

CHAPTER 3

THE TABLEAU VM SCHEDULER

In this chapter, we first look at the high-level role of a VM scheduler and the design decisions that affect the runtime performance of client VMs. In particular, we look at two key sources of performance volatility and degradation: the use of heuristics during the scheduler's decision-making process and high runtime overheads of scheduling operations. Based on this, we present the high-level design of Tableau, a novel approach to VM scheduling that is designed from the ground up to avoid these pitfalls.

3.1 The Role of a VM Scheduler

While a VM in modern hypervisors, by itself, serves as a container for various resources (*e.g.*, CPU, memory, I/O devices and bandwidth), from the perspective of a VM scheduler, a VM can be simplified to be a container for one or more *virtual CPUs* (or vCPUs). The role of the VM scheduler, at a high level, is to ensure that all vCPU's in the system receive a user-configured share of CPU time, while minimizing the latency and maximizing the throughput of tenant applications. At a lower level, this means that each vCPU must be allowed to execute for a certain cumulative share of CPU cycles and these cycles must be provided in a way so as to ensure the incurred *scheduling latency* stays within acceptable bounds.

It should be noted that the definition of "acceptable bounds" can vary significantly across current public cloud providers, as well as across different tiers of VMs from the same provider. Regardless, we assume the presence of some quantitative value of scheduling latency, whether it be through implicit means (*e.g.*, based on customer expectations as a result of advertising) or via explicit means (*e.g.*, a *service-level agreements* (SLAs) signed between customers and providers).

Scheduling Latency and Maximum Scheduling Latency: We define the *scheduling latency* incurred by a particular vCPU as the delay introduced by the scheduler between consecutive timeslices during which the vCPU is executing. This occurs in all schedulers when either **(i)** multiple uncapped vCPUs contend for a single core or **(ii)** the maximum CPU share of a vCPU is capped to less than a single core. Controlling this scheduling latency is crucial as it is visible to clients in the form of increased application latencies. We define the *maximum scheduling latency* as the maximum-observed scheduling latency from the point of view of a virtualized application.

The goals of a VM scheduler, namely ensuring a certain CPU share and guaranteeing a bound on application latency for every VM in the system, are affected by two primary factors: **(i)** increased and unpredictable tail latencies for tenant applications resulting from *unpredictable scheduling heuristics*, as employed by many popular VM schedulers, and **(ii)** lowered application throughput due to *high scheduler runtime overheads*, which eat up precious CPU cycles that could have been used by tenant VMs. We expand on these two factors in more detail below before providing an overview of Tableau's high-level design.

3.1.1 Heuristics Increase Tail Latencies

To improve the average-case latency of VMs, many VM schedulers employ heuristics and special-case optimizations that favor VMs performing I/O [25, 93, 107].

For example, when a vCPU resumes (or *unblocks*) from a blocking I/O operation, Xen's *Credit* scheduler attempts to temporarily "boost" its priority, thereby overriding the fairness criterion. Such heuristics are not unique to Xen and are commonly implemented in most major cloud hypervisors today. For example Linux's *Completely Fair Scheduler* (CFS), which is widely used in conjunction with Linux's built-in KVM hypervisor, also uses accounting tricks to favor I/O activity (*e.g.*, the "gentle fair sleepers" setting [93]). In fact, CFS has even been observed to under-utilize cores in fully loaded systems due to complex, erratic, and often erroneous load-balancing heuristics [74]. Similarly, in the VMware ESXi scheduler, a VM "that is frequently blocked due to I/O gets scheduled more promptly compared to [a VM] that is CPU bound if all other conditions are equal." [107].

The problem is that while such approaches result in improved average-case scheduling latencies (and consequently, application latencies) for I/O-bound VMs, as we show later in Chapter 6 for Xen, it also induces unpredictable, hard-to-anticipate delays in

scheduler tail latencies. For example, in the case of the Credit scheduler in Xen, if every vCPU is performing I/O and is boosted to a higher priority, then effectively no vCPU is boosted since they all contend at the same higher priority level. Combined with the multi-tenant nature of public clouds, which host a diverse range of workloads from different customers with varying runtime characteristics, it is difficult to predict what the observed tail latencies will be for a given vCPU at any given time.

A secondary, but important, concern is that in order to implement such a heuristic, one must introduce complex runtime logic into the VM scheduler to keep track of metrics and perform various book-keeping operations that aid the decision-making process. This results in CPU cycles being spent on work that could have been used by tenants. For example, in Xen's Credit scheduler, a periodic task tracks vCPU runtime statistics in order to characterize their I/O sensitivity. It also requires more complex dispatch logic, a critical system hotpath, which makes use of this information.

3.1.2 Scheduling Overheads Limit Throughput

A second major concern is that a VM scheduler must exhibit very low runtime overheads. There are two reasons for this requirement: first the scheduler is frequently invoked and any wasted cycles during key runtime operations are multiplied as a result. Second, because any cycles spent on scheduling are pure overhead in that they would otherwise have been available to applications of paying customers. Therefore, it can be argued that any cycles used beyond the minimum requirements of satisfying a VM's SLA are detrimental towards the performance of VMs and the useful work performed on the machine as a whole. To reiterate the previous example regarding the use of heuristics, in addition to causing unpredictable latency spikes, dynamic scheduling heuristics are also detrimental to throughput because they must be frequently computed. Similarly, any other major source of runtime overheads such as lock contention inside the scheduler hurts application throughput.

At cloud scale, such overheads can accumulate towards massive costs. For example, a recent study of more than 20,000 computers in one of Google's data centers found that roughly 5% of all processor cycles are spent on the kernel's process scheduler [58]. In the case of high-density workloads, the effects of any scheduling bottlenecks are further exacerbated by the fact that the rate at which context switches occur is naturally higher.

There is thus strong motivation to make the VM scheduler as efficient as possible. However, as we demonstrate in our evaluation, contemporary VM schedulers leave substantial room for improvements in terms of both runtime overheads and scheduler-induced tail latencies when facing challenging high-density workloads that frequently

trigger the VM scheduler. As a novel, unorthodox alternative that occupies a previously unexplored point in the design space of VM schedulers, we propose Tableau, a low-overhead VM scheduler that *guarantees* a minimum share of CPU time and a hard bound on maximum scheduling latency for every vCPU.

3.2 Design Overview

Tableau is based on a table-driven design inspired by hard real-time systems to minimize runtime overheads while maintaining high throughput and predictable latencies even when confronted with a large number of VMs. In the following, we introduce the high-level design; implementation-level details are discussed in Chapter 4.

3.2.1 Dispatcher vs. Planner

VMs in cloud environments are typically long-running. For example, a majority of customer VMs hosted on Microsoft Azure have a lifetime of at least a few hours [29], and VMs that run longer than a day are likely to run for several days and account for more than 95% of the total core hours [29]. Based on this key observation, we push all expensive scheduling logic related to satisfying VM performance requirements into a separate, infrequent planning (or *system reconfiguration*) step that is only invoked when a VM is started up, torn down, or reconfigured.

Consequently, Tableau's scheduler consists of two main components: a straightforward, low-overhead, table-driven *dispatcher* and a relatively heavy-weight *scheduling table generator* (or *planner*). The dispatcher resides in the hypervisor and is invoked whenever a scheduling decision is needed. It simply enacts the latest scheduling table provided by the planner. The planner in turn can reside anywhere (*e.g.*, it can be an unprivileged process) and is invoked if a new table is needed, *i.e.*, when a system reconfiguration occurs. By leveraging multiprocessor real-time scheduling theory (discussed in Chapter 5), the planner *quickly* generates tables that *guarantee* a minimum CPU share and a hard upper bound on the maximum scheduling delay for each VM in the system.

An immediate benefit of the split between a minimal, efficient dispatcher and a separate system-wide planning process is that the dispatcher uses *primarily core-local data structures*, which trivially ensures Tableau's scalability on large multicore platforms. All decisions requiring system-wide information and coordination as well as table updates are done asynchronously by the planner and do not slow down the online dispatcher.

Thus, the performance-critical scheduler hot paths are not impacted by the planner's overheads.

3.2.2 Second-Level Scheduler

A naive table-driven scheduler, however, is too inflexible at runtime and results in non-work-conserving behavior. As a result, the average scheduling latency incurred by VMs is higher due to the *blackout periods* that occur between consecutive slots in the scheduling plan. That is, if an I/O request for a VM arrives between two of its slots in the current plan, the I/O request is deferred *at least until the beginning of the next slot*, after which it will incur the usual intra-VM queueing delays before being serviced. Therefore, a table-driven scheduler alone has the downside of observably higher average-case I/O latencies for applications. Further, it might be the case that the CPU ends up being idle because the VM whose slot is currently active is idle.

Ideally, we want to be able to utilize these idle cycles, whether they result from inter-slot idle periods or intra-slot idle time due to VMs idling. To remedy this, we introduce a second-level scheduler that is invoked whenever the table-driven dispatcher picks the idle vCPU (*i.e.*, when it does not have any valid table-driven VM to schedule).

The second-level scheduler then picks, in a round-robin fashion, one of the other table-driven VMs assigned to that core that are runnable. Therefore, a table-driven VM that receives an I/O request during an idle period, and becomes runnable, ends up processing the request immediately when dispatched by the second-level scheduler.

Since being scheduled by the second-level scheduler results in VMs receiving more CPU time than agreed upon in the SLA, we make the distinction in Tableau between *capped* and *uncapped* VMs (similar the other hypervisor schedulers). Capped VMs are restricted to executing within their slots specified in the current plan, and incur the higher average-case latency. This is similar to other VM schedulers like Xen's Credit scheduler, where a capped VM is cut off from further CPU time regardless of whether idle cycles are available in the system. On the other hand, uncapped VMs are guaranteed the time within their slots in the current plan but are allowed to execute during idle periods via the second-level scheduler.

Another way to think about this is that capped VMs are guaranteed a fixed share of CPU time, while uncapped VMs are guaranteed a *minimum* share of CPU time, with no guarantees for CPU time beyond that. In both cases, Tableau guarantees scheduler latency bounds.

3.2.3 Background Scheduler

Finally, since it is generally the goal of cloud operators to use each and every available cycle to maximize profits, it is advantageous to provide a way to harvest any idle cycles that are left after both the first- and second-level scheduler have attempted to schedule table-driven VMs.

To accommodate this, Tableau allows for a third group of VMs to exist in the system, which are scheduled by a third-level *background scheduler* using a simple round-robin fair-share algorithm. The background scheduler is invoked only when both the first- and second-level schedulers fail to find a vCPU to schedule and consequently only harvests idle cycles that would have otherwise gone to waste.

Therefore, in addition to uncapped and capped VMs with strong guarantees on CPU share and scheduling latency, the background scheduler enables a low-priority, best-effort class of VMs to co-exist on the same machine without affecting the utilization and latency characteristics of client VMs. These low-priority VMs simply soak up any unused cycles in the system and can be used for performing other useful work. For example, cloud providers can choose to provide these idle cycles for use internally within their organization to run secondary, CPU-intensive workloads such as an extension of search indexing or for fuzzing product APIs to improve their security.

Finally, providers can also simply choose to rent these VMs out to customers at a significantly lower cost owing to the lack of strong performance guarantees in the default case. That said, it should be noted that performance guarantees for background VMs can be trivially introduced by limiting the utilization per core and ensuring the presence of a minimum amount of idle time during the planning step. In doing so, the configured idle fraction of each core will be divided among the background VMs, providing a lower bound on their average, long-term performance.

Tier-1 vs. tier-2 VMs: To accommodate the discussion of the various levels of the Tableau scheduling approach, we differentiate the terminology between two types of VMs for the rest of this thesis. We refer to VMs that have performance guarantees and are scheduled via the table-driven dispatcher and second-level round-robin scheduler as *tier-1* VMs. On the other hand, we refer to low-priority VMs scheduled via the background scheduler as *tier-2* VMs. We also make the distinction between *capped* and *uncapped* tier-1 VMs: capped tier-1 VMs execute only via the table-driven dispatcher, while uncapped tier-1 VMs additionally get dispatched by the second-level scheduler. Finally, tier-2 VMs are always assumed to be uncapped and, apart from being subject to limitations imposed on them by the background scheduler itself, have no limits on how much idle time they can soak up.

To summarize, Tableau is a three-level, hierarchical scheduling approach aimed at cloud providers with a table-driven dispatcher at the first level, a core-local fair-share scheduler at the second level to enable uncapped VMs, a background scheduler for running low-priority workloads, and an infrequently invoked asynchronous planner. Together, these components ensure flexible, work-conserving runtime behavior on top of the minimum performance guarantees incorporated into the tables, which are (re-)generated on demand.

We next elaborate on the dispatcher in Chapter 4 and then discuss how the planner finds scheduling tables with performance guarantees in Chapter 5.

CHAPTER 4

DISPATCHER IMPLEMENTATION

This chapter gives a detailed description of our Xen-based implementation of the table-driven dispatcher described in the previous chapter, as well as an overview of the second-level and background schedulers in Tableau.

4.1 Table-Driven Dispatcher

At the first and highest level, Tableau schedules tier-1 vCPUs using a table-driven dispatcher, not unlike those commonly found in safety-critical hard real-time systems. For instance, the ARINC 653 standard for *integrated modular avionics* (see Section 2.5.4) specifies time-partitioned scheduling, which is accomplished with static scheduling tables. We adopt this proven technique for building a highly predictable VM scheduler.

A table-driven dispatcher requires a pre-generated scheduling table of finite length (usually in the range of a few hundred milliseconds). For each core, the table is provided as a set of non-overlapping intervals, each of which is specified using offsets relative to the start of the table. Each interval is either marked as idle or reserved for a specific tier-1 vCPU that is given the highest priority during that interval

When the dispatcher is invoked at runtime, it simply looks up the interval in the scheduling table corresponding to the current system time (modulo the table length). If this interval is reserved for a specific vCPU, and if that vCPU is ready, it is dispatched and allowed to run uninterruptedly until the end of the current interval, or until it blocks, whichever comes first.

If the specific vCPU is blocked (*e.g.*, while waiting for an I/O request to complete), or if the current interval is marked as idle, the second-level scheduler is invoked to schedule any ready *core-local* tier-1 vCPUs, which are chosen in a round-robin manner.

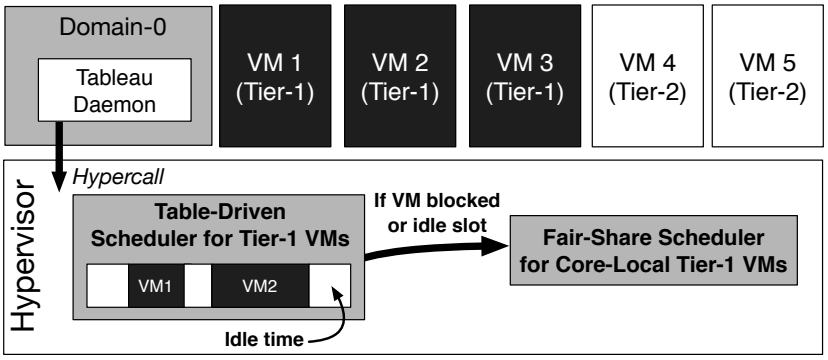


FIGURE 4.1: Tableau architecture

To summarize, the scheduler hotpath in Tableau consists of little more than a straightforward table lookup in the common case, which is a minimal and hence extremely efficient approach to scheduling. The schedule resulting from the table repeats cyclically until a new table is installed by the planner. Importantly, as we show later in Chapter 5, it can be made inherently predictable: the maximum "blackout time," during which a vCPU receives no service, and which directly translates into application-visible latency, can be trivially bounded by applying techniques from multiprocessor real-time scheduling theory during table generation. It is also work-conserving (w.r.t. core-local vCPUs) owing to the second-level scheduler.

The Tableau approach is not tied to any particular system and can be realized in virtually any modern hypervisor. For evaluation purposes, we chose the popular Xen hypervisor (version 4.9) as the basis for our experiments, due to its widespread use in public clouds such as Amazon EC2 at the time of our implementation efforts.

The main components of Tableau in Xen are illustrated in Figure 4.1. Recall that Xen consists of a special supervisory VM called *domain-0*, or *dom0*, which has privileged access to the underlying hardware to enable **(i)** device access, and **(ii)** the creation, teardown, and reconfiguration of domains. Accordingly, the planner is realized as a daemon in the userspace of dom0 (henceforth referred to simply as userspace).

Implementing the scheduling logic in userspace is quite convenient. In particular, the Tableau planner is written in Python using *SchedCAT*, an open-source real-time scheduling toolkit [1]. The use of a high-level language greatly simplifies the rapid exploration of new post-processing phases and scheduling ideas, potentially even by non-systems developers or using machine-learning techniques.

In total, our Tableau prototype consists of around 2,350 lines of new or changed C code in the hypervisor itself, and around 1,600 lines of code in the userspace Tableau daemon. It was possible to realize Tableau with a relatively small code base because the hypervisor component is simple by design, and because the planner heavily relies

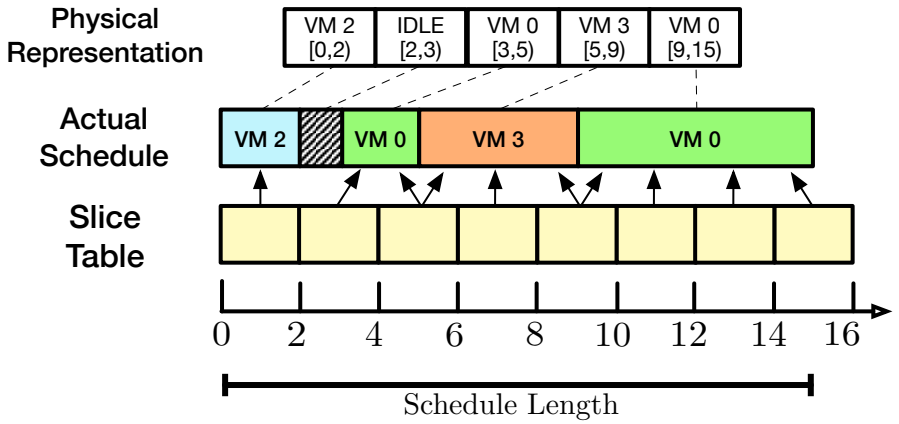


FIGURE 4.2: Scheduling table and slice table

on existing scheduling logic [18] in SchedCAT.

New scheduling tables are pushed by the planner to the hypervisor via a hypercall in a compiled, binary format and used directly by the Tableau dispatcher. While the dispatcher is conceptually straightforward, there are certain choices involved in implementing it efficiently. In the following, we highlight four key aspects.

4.2 Tier-1 Scheduling

In this section we look at the first- and second-level scheduler used to scheduler tier-1 VMs. We first look at the algorithm used by the tier-1 scheduler before giving a brief overview of the round-robin approach used in the second-level scheduler.

4.2.1 Table-Driven Scheduler

Figure 4.2 shows the structure of a Tableau scheduling table as it is provided by the planner. It consists of per-CPU lists of allocations, which map an interval of time to a specific vCPU.

The first implementation-level challenge that we must address is to design an efficient mechanism to quickly lookup the current slot in the scheduling table. The problem is that an allocation in the table represents a variable-length interval within the table. In the worst-case (*i.e.*, we are in the last slot of the table) we have to iterate through each slot in the linked list until we come to the currently-active slot. Using a more compact structure with a fixed-size array might improve the lookup to $O(\log n)$, but is still not constant time.

$O(1)$ lookups using a slice table. To facilitate constant-time lookups, the scheduling table is accompanied by a *slice table*. A slice table is essentially an index comprised of "slices" of the allocation table, where each slice describes a *fixed-sized time interval* of the allocation table. The slice length is chosen such that each slice overlaps with at most two allocations (and possibly some idle time between them). This is accomplished by picking, for each pCPU, a per-CPU slice length equal to the length of the shortest allocation on that particular pCPU. Effectively, the inclusion of the slice table allows us to translate from the time domain to a specific slot within the table in constant time.

The slice table therefore enables $O(1)$ scheduling decisions. First, the dispatcher determines the current slice by indexing the slice table using the current time (modulo the table length), and then it determines which of the two allocations within the slice (or the idle time between them) currently needs to be scheduled. The allocation and slice-table records are aligned to cache lines, so at most two cache lines are accessed per lookup.

The simple table-driven dispatcher is efficient and inherently scalable as most memory accesses are to core-local data structures, especially in the common-case hot path. However, in two exceptions, which we briefly sketch next, remote accesses are needed.

Cross-core migrations. In Tableau, as we will see later in Chapter 5, a vCPU may have allocations on two cores, due to semi-partitioning, if that is the only way it can be scheduled. One challenge is that if the gap between these two allocations in the table is small (or even overlapping by a few cycles due to timer skew), we must ensure that one core does not schedule the vCPU until it has been completely de-scheduled on the other core (to avoid stack corruption).

To this end, for each vCPU, Tableau tracks the core that currently schedules the vCPU, if any. Before scheduling a vCPU, a core checks that it "owns" the vCPU. If the vCPU is still marked as "scheduled elsewhere," the core that failed to schedule the vCPU sets a field in the vCPU structure requesting an *inter-processor interrupt* (IPI) to be sent when the vCPU is de-scheduled, and schedules either a vCPU selected by the second-level scheduler or idles until notified. No locks or cache lines shared by all pCPUs are required.

In the expected case (no overlap of allocations), the only cost is an atomic write to the vCPU control block (which is already in-cache anyway). In the rare case of a race between allocation start and end times, a remote memory reference and an IPI are occasionally incurred, which however does not impact scalability.

Blocking vCPUs. Blocking a vCPU does not involve any action since the state of each vCPU is checked and ensured to be runnable before attempting to schedule it. Therefore, when a slot switches to that of a blocked vCPU, the scheduler will simply see that the state is not runnable and choose to invoke the second-level scheduler.

Unblocking vCPUs. Tableau must also deal with wake-ups of blocked vCPUs, which may be processed on any core in the system. For each vCPU, we keep track of the core it currently has an allocation on (or where it last had an allocation). When a core processes a wake-up for a vCPU that has a current allocation, it reads this field and sends an IPI to the responsible core. Similarly, if the vCPU does not have a current allocation, but is allowed to take part in second-level scheduling, and the vCPU's last-used core is currently idling, then an IPI is sent to said core.

If, however, the vCPU does not currently have an allocation on any core and is capped (*i.e.*, not eligible to take part in second-level scheduling), then the wake-up can be safely ignored; when the next allocation pertaining to the vCPU begins, it will be seen to be runnable anyway. Again, no locks or globally shared cache lines are required to realize this optimization.

For simplicity, it is also possible to unconditionally send an IPI; if IPIs are relatively cheap, then this may be preferable to complicating the wake-up logic. Our prototype currently uses this approach and simply unconditionally sends an IPI whenever a vCPU unblocks.

Synchronization-free table switching. To avoid adding a lock or a barrier in a hot path, table switches in Tableau are time-synchronized. Each core has a $next\_table$ pointer, which is set when a new table is pushed. If a core finds this field to be set when the current table wraps around, then it switches to the new table (otherwise the current one is reused). However, if the $next\_table$ pointer is set *during* a table wrap, some cores may pick up the change while others may retain the old table, causing an inconsistent schedule. To avoid such races, we simply ensure that tables are never set during or close to a table wrap. When a new table is pushed, all $next\_table$ pointers are timed to be set at a point in the middle of the next round of the current table. Given the scheduling table length in Tableau ($\approx 102\,ms$), this technique avoids any race and all cores consistently switch to the new table. Two rounds after a new table has been uploaded, when all cores are certain to have switched to the new table, the previous table is garbage-collected.

4.2.2 Second-Level Scheduler

The second-level scheduler is a simple round-robin scheduler that is invoked when the table-driven dispatcher idles. This may occur for one of two reasons: either the current slot is an idle slot, or the current slot is non-idle but the vCPU to be scheduled is currently blocked on I/O.

The second-level scheduler works within a configurable periodic interval called an *epoch*, which is set to a default of 20 ms. At the beginning of every epoch, a *budget replenishment* occurs where each currently runnable VM is assigned an equal share of the next epoch. For example, if there are currently four runnable tier-1 VMs on a core with a 20 ms epoch, each of them is assigned a 5 ms budget for the upcoming epoch. If a new VM wakes up during that epoch, it will be accommodated in the next epoch by assigning each VM 4 ms (now that there are five ready VMs). If a VM that was assigned a budget in the current epoch blocks during the epoch, it maintains its budget for that epoch. Therefore, if it were to unblock at some point in the future during the same epoch, it would resume execution with the budget it had prior to blocking.

When invoked, the second-level scheduler iterates through the list of ready tier-1 VMs on the current core, and picks the first one with sufficient budget and schedules it. It also sets up a timer to fire in the future when the budget of the vCPU expires. If the vCPU executes without blocking, this timer fires at the point where the vCPU's budget would have expired. Within the timer handler, the budget of the scheduled tier-1 vCPU is deducted by calculating the total time difference from the current time to the time when the vCPU was previously dispatched.

Semi-partitioned VMs. The current implementation of the second-level scheduler only considers tier-1 VMs that are not semi-partitioned (*i.e.*, only those vCPUs that have slots on a single core are considered), which means that semi-partitioned VMs cannot currently make use of spare idle cycles.

The underlying reasoning for this choice is that semi-partitioned VMs, owing to their having slots on different cores, require additional synchronization to ensure that second-level schedulers on multiple cores do not attempt to schedule them simultaneously. For example, if a VM is semi-partitioned across cores 1 and 2, both of which are currently idle, we must incorporate additional logic in the scheduler hotpath to ensure that they do not both end up scheduling the semi-partitioned VM simultaneously, which would result in memory corruption as both cores attempt to use the VM's stack area.

This problem is not unique to the second-level scheduler and is handled in the table-driven scheduler as discussed above in Section 4.2.1 (see discussion on "cross-core migrations"). However, synchronizing cross-core migration in a table-driven scheduler is significantly easier as the table provides complete information regarding where a VM might be currently executing. In the case of the second-level scheduler, one would incur higher synchronization overheads as it would need to search through the ready queues of all second-level schedulers on all cores a VM may currently be running on.

Since attempting to introduce additional synchronization into the second-level scheduler hotpath would affect all VMs, and because semi-partitioned VMs are expected to be absent in most production scenarios (or present only in extremely rare circumstances), we simply ignore them for second-level scheduling. The alternative would be to increase the cost of the second-level scheduling hotpath in favor of a rare case (*i.e.*, uncapped semi-partitioned VMs), at the expense of the common case (uncapped fully-partitioned VMs).

It should be emphasized that ignoring semi-partitioned VMs at the second-level scheduler has no effect on the performance guarantees provided by the first-level scheduler: semi-partitioned VMs continue to enjoy the guarantees provided to them via the current scheduling table. They simply cannot be uncapped, and thus cannot use up additional idle cycles in the system.

4.3 Tier-2 Scheduling

The Tier-2 scheduler, or the background scheduler, is responsible for dispatching tier-2 VMs whenever the first two scheduling levels idle because they cannot find any VM to schedule.

The background scheduler has a design nearly identical to the second-level scheduler: background VMs are partitioned onto individual cores upon creation and, on each core, the background scheduler employs a round-robin approach to schedule any runnable tier-2 VMs within a configurable periodic epoch.

Dealing with non-work-conservation. Unfortunately, idle time may not be distributed evenly across cores in the system, and depends on the idling behavior of both capped and uncapped tier-1 VMs. This means there may be situations where idle time is available on a core but due to a prior partitioning, it does not have any tier-2 VMs to run. At the same time, tier-2 VMs may be starving on a different core due to a sudden burst of tier-1 work. This non-work-conservation may be acceptable to some extent given the

lower priority of tier-2 VMs, however, a prolonged periodic of non-work-conserving behavior is undesirable. Rather, we require some form of load balancing to mitigate this.

One way to ensure work-conservation is to incorporate a more complex global scheduler design for tier-2 scheduling that load balances tier-2 VMs across all cores in the system as they become idle. However, we eschew this approach as it would require additional cross-core synchronization that introduces runtime overheads that affect the first- and second-level schedulers.

The key difference in design between the second-level and background schedulers is that the background scheduler keeps track of idle-time statistics for each core in the system. It tracks the total cycles that were idle in the last n rounds of the table, where n is a configurable parameter. This idle-time is then extracted (free of any synchronization) by a userspace load-balancing daemon that periodically re-partitions tier-2 VMs, accounting for any changes in the distribution of idle time in the system (averaged across a configurable number of scheduling table cycles).

Currently, the repartitioning metric is simple and just assigns tier-2 VMs to cores in proportion to the average idle time they experienced in the last n rounds. The repartitioning period of the daemon can be configured depending on how much non-work-conserving behavior one is willing to tolerate, with the trade-off being increased CPU usage for dom0 due to the frequency of actions of the load-balancing daemon. A high frequency reduces the delay during which tier-2 scheduling is non-work-conserving, but increases the CPU usage of the load-balancing daemon. On the other hand, a lower load-balancing frequency reduces CPU usage but increases the potential for temporary starvation of tier-2 VMs due to the non-work-conserving behavior of the tier-2 scheduler.

Finally, we note that when a VM is created, it is initially always created as a tier-2 VM. This means that all VMs, including VMs that will eventually be *promoted* to a tier-1 VM (via a later table push) begin their lifecycle as tier-2 VMs. The advantage of doing so for tier-1 VMs is that they begin executing immediately if there are any idle cycles available. On the other hand, there is a slight delay between when a tier-1 VM is created as a tier-2 VM initially and when a table push promotes it to a tier-1 VM. However, since table re-generation is invoked as soon as a VM destined to be a tier-1 VM is created, this delay is predictably bounded: it takes at most the overhead of a table push plus the time taken for a single round of the planner to complete. As we show later in Chapter 6, this delay is typically a fraction of the time taken for a VM to boot up, and so we consider it acceptable.

4.4 Summary

To summarize, Tableau is implemented via a three-level hierarchical scheduler. The first level schedules tier-1 VMs strictly using a table provided to it by the planner. The second-level additionally enables runnable, uncapped tier-1 VMs to use up any idle cycles available in the system when the first-level scheduler idles. Finally, a third-level background scheduler soaks up any remaining idle cycles in the system and gives them to tier-2 VMs, a lower-priority set of VMs. The background scheduler uses a partitioned scheduling approach that is susceptible to non-work-conserving behavior, and to remedy this, a userspace load-balancing daemon extracts idle-time statistics gathered by the background scheduler to periodically re-partition tier-2 VMs. This allows for maintaining the advantageous properties of a partitioned scheduler (*i.e.*, simplicity, low overhead) while providing approximate work conservation.

CHAPTER 5

PLANNER DESIGN

In this chapter we detail the *planner* component of Tableau. In particular, we look at the approach used by the planner for generating good scheduling tables for a given system configuration.

From the planner's perspective, the view of the system consists of a list of vCPUs, each with some configuration parameters. More specifically, the planner assumes (and requires) a specified *reserved utilization* U and a *maximum scheduling latency* L for each vCPU in the system, which may be selected by system administrators based on their requirements. For example, they may be explicitly specified by an associated SLA, predetermined according to price-differentiated service tiers set by the cloud provider, or simply computed by a fair-share policy (*e.g.*, $U = \frac{m}{n}$, where m is the number of CPU cores and n the number of vCPUs assigned to the host).

Crucial to note is that Tableau does *not* require more information to be provided than existing fair-share schedulers such as Xen's Credit scheduler or Linux's CFS scheduler. Just as in Xen's Credit scheduler or KVM's CFS, U can be determined automatically based on a vCPU- or VM-specific weight, the number of cores, and the number of vCPUs in the system. Similarly, L can be given a reasonable default magnitude similar to the scheduling quantum in Credit or the `sched_latency_ns` tunable of CFS.

The advantage of Tableau is that it *additionally* allows for more sophisticated performance- or price-differentiated provisioning strategies without the added administrative complexity of a more complicated default setup or a higher barrier to adoption.

Based on this, the challenge for the planner is to find a static vCPU schedule for the dispatcher that is runtime-efficient and that satisfies the minimum utilization and maximum latency guarantees for all vCPUs in the system.

Since the planner can operate outside the restricted confines of the hypervisor, such as within a supervisory VM, one might be tempted to use high-level tools such as ILP

or SMT solvers to find schedules. However, we want the table generation process to be relatively fast (*i.e.*, seconds rather than minutes) even for hundreds of vCPUs and therefore avoid such heavyweight solutions.

Rather, we map the problem to the well-studied problem of multiprocessor hard real-time scheduling, which allows us to quickly generate reasonably short tables satisfying all constraints for *any possible configuration* of VMs that does not over-utilize the system (*i.e.*, where the sum of all U parameters does not exceed the number of available cores).

The planner generates tables in two steps. It first models each vCPU as a *periodic task* [72] with parameters that are carefully chosen to **(i)** reflect its specified U and L parameters while **(ii)** ensuring a short maximum table length. It then simulates a multiprocessor real-time schedule of the set of periodic tasks representing all vCPUs in the system. This simulation results in a repeating table that *guarantees* the target utilization and ensures the desired scheduling latency for each vCPU.

We now describe in detail each of the various steps taken by the planner to create a scheduling table.

5.1 Mapping vCPUs to Periodic Tasks

The first step performed by the planner is to model each vCPU in the system as a periodic task. Recall that a periodic real-time task [72] $\tau = (C, T)$ is characterized by its *worst-case execution time* C and *period* T. That is, a periodic task is assumed to *release* a job every T time units, with each job taking *at most* C time units anywhere during the current period's interval. The only associated correctness criterion is that each job released by a periodic task must receive (up to) C time units of processor service from the scheduling during each scheduling interval $[0, T)$, $[T, 2T)$, $[2T, 3T)$, *etc.*

When mapping a vCPU (U, L) to a periodic task $\tau = (C, T)$, we clearly require $U = \frac{C}{T}$. However, while we know the ratio between C and T, how do we map a vCPU's latency goal L to an "equivalent" period T?

Without knowing anything about the final schedule, a suitable period can be determined by observing that a periodic task must be scheduled for at least C time units during every period of length T. The worst-case blackout time (*i.e.*, contiguous interval without any processor service) hence occurs when a periodic task is scheduled for C time units at the very beginning of one period, and then scheduled next only at the very end of the next period, again for C time units. For example, a periodic task with

$(C, T) = (10ms, 100ms)$ might be scheduled during $[0ms, 10ms)$ and then again during $[190ms, 200ms)$, yielding a blackout time of $180ms$ corresponding to the blackout interval $[10ms, 190ms)$.

In general, the worst-case blackout time incurred by a periodic task with period T and cost C is bounded by $2 \times (T - C)$, or equivalently $2 \times (1 - U) \times T$. Thus, a vCPU's latency goal L can be converted into a suitable period T by picking any period T such that $T \leq \frac{L}{2\times(1-U)}$. If $U = 1$, then the vCPU is simply mapped to a dedicated pCPU and excluded from further consideration.

Bounding table lengths. While the simple approach above results in periodic tasks that satisfy the U and L associated with each vCPU in the system, in order to minimize preemptions, one should attempt to maximize the period (*i.e.*, a shorter period results in more frequent preemptions that degrades performance due to system overheads and cache-related preemption delays).

However, simply choosing the maximal period for each vCPU can result in a periodic task set with an extremely large *hyperperiod*, the least common multiple of all task periods. Since this is also the length at which the dispatching table repeats, picking periods indiscriminately could even result in exponential table sizes (if all chosen periods are relatively prime). This would result in significant memory usage as well as significant cache usage by the frequently-invoked scheduler hotpath.

To avoid this problem of ending up with potentially large dispatching tables, we instead select periods from a set of candidate periods with a known maximum hyperperiod. Specifically, in our implementation, we searched for a number close to 100 ms (=100,000,000 ns) that has a large number of factors larger than 100μs (since periods smaller than 100μs are hard to enforce due to scheduling overheads). We chose 102,702,600 ns as the maximum hyperperiod, which has a large number of integer divisors (186) above the 100μs threshold.

With this simple extension the planner can pick the best period for a particular vCPU from any of these 186 divisors. In our planner implementation, we choose the nearest candidate period that is less than the computed maximal period, without facing the risk of resulting in large table lengths. More precisely, if F denotes the set of all integer divisors of 102,702,600 greater than 100,000, we select for each vCPU the largest $T \in F$ such that $2 \times (1 - U) \times T \leq L$. Depending on the chosen T, tenants may observe less scheduling delay than stipulated by L, which is consistent with it being an *upper bound* on scheduling latency.

VM	U	L
vCPU_1	0.60	25
vCPU_2	0.60	50
vCPU_3	0.60	100

TABLE 5.1: Example vCPUs corresponding to three VMs, each with a utilization U and maximum scheduling latency L.

Task	C	T
τ_1	18	30
τ_2	36	60
τ_3	72	120

TABLE 5.2: The equivalent periodic tasks corresponding to the vCPUs shown in Table 5.1 with a budget C and a period T.

Choice of Table Length: The choice of table length is arbitrary, but guided by certain principles: the length of approximately $102\,ms$ is short enough to be generated and replaced quickly, and has a large number of factors that form the set of candidate periods, and allow for the final period to be as close to the requested one as possible. In general, a shorter table can be made longer if needed by simply unrolling it multiple times. For example, if there is only a single VM on a core, it might make sense to unroll the table as much as possible to avoid the slightly increased overhead of table wrap-arounds (owing to fact that an additional branch is needed to check if the table has been switched during the last round). This of course has the downside of increasing the time taken for table switches to take effect, as the new table would only take effect a minimum of one entire cycle later.

Running example. Consider the simple example workload in Table 5.1. Each VM has a target utilization of $U = 60\%$, but differing maximum acceptable scheduling delay bounds of $L \in \{25ms, 50ms, 100ms\}$, which for example could reflect three price- and service-differentiated instance types with varying scheduling delay guarantees.

In the case of vCPU_1, we observe that the largest candidate period T in F satisfying $2 \times (1 - 0.60) \times T \leq 25ms$ is $T = 30$ms. Hence, the task τ_1 corresponding to vCPU_1 is given a period $T = 30$ms, which implies a cost of $C = U \times T = 18$ms. Similar reasoning for the other vCPUs yields the periodic tasks listed in Table 5.2.

Once each vCPU is represented as a periodic task, the planner must find a schedule that satisfies the timing constraints of all periodic tasks, in which case all vCPU utilization and latency goals are guaranteed to be met. To this end, Tableau uses a progression of three increasingly expensive techniques: first a very simple and quick bin-packing heuristic that we expect to be sufficient in most practical use cases, and then two more involved scheduling techniques that we include primarily for the sake of completeness (*i.e.*, to ensure that the planner never fails, even in pathological scenarios).

5.2 Partitioning

We begin by attempting to *partition* the task set, that is, to statically assign tasks to individual cores such that no core is overloaded. Such an approach is a desirable first step as it results in high cache affinity (since no vCPUs migrate between cores). Partitioning also has the advantage that additional considerations can be easily incorporated. For example, memory locality on NUMA platforms can be enforced by limiting the set of CPUs that a vCPU can be partitioned on to the subset associated with the NUMA domain its memory resides in. Similarly, special treatment of hardware threads (*e.g.*, to avoid side channel attacks) can be incorporated by preventing vCPUs of different VMs from being partitioned onto two threads of the same core. Finally, cache interference concerns can be mitigated to some extent by partitioning memory-intensive vCPUs onto different sockets (with a distinct L3 cache), or if the working set is smaller and fits in L1 or L2 cache, onto different cores.

Partitioning periodic tasks onto cores is a bin-packing-like problem that is NP-hard. We use the well-known *worst-fit decreasing* heuristic (always assign the next task to the least-utilized core), which has the benefit that it distributes the load roughly evenly across all cores in the system.

Choice of Partitioning Heuristic: While we use the worst-fit decreasing heuristic for partitioning vCPUs onto cores, others may be used in order to achieve different properties. The worst-fit decreasing heuristic has the advantage of spreading vCPUs onto as many cores as possible, thereby lowering intra-core interference among vCPUs partitioned on the same core. For example, partitioning sixty-four VMs of 25% utilization each across thirty-two cores using the worst-fit decreasing heuristic would result in each core being assigned two VMs. This is in contrast to a best-fit decreasing heuristic that would pack VMs tightly, resulting in only sixteen cores being assigned four VMs each. Naturally, other situations might require optimizing for different concerns. For example, a best-fit decreasing heuristic as shown in the example above can be used to instead pack vCPUs onto as few cores as possible, allowing for powering down entire cores. This can be a useful alternative if minimizing power consumption is a primary goal.

If the partitioning heuristic succeeds in finding a valid partition, we simply simulate on each core an *earliest-deadline-first* (EDF) schedule until the hyperperiod, keeping track of the scheduling intervals and the vCPU being scheduled. Since EDF is optimal on

uniprocessors [72], the simulation guarantees a schedule satisfying all utilization and latency goals.

Running example. Recall the task in Table 5.2, and suppose we have two cores. Since all tasks have the same utilization, it does not matter in which order they are considered. The first two tasks can be trivially assigned to the two cores. However, there is no way to fit the third task without overloading a core. The heuristic thus fails.

It is important to note that although the example above is designed to illustrate a case where partitioning fails, it is expected to succeed for all practical applications of Tableau. The primary use case for Tableau is to support VMs on machines in datacenters run by a cloud provider. It is not only practical, but desirable, to segment VM offerings into a few configurations. This is because providing clients with fine-grained control over low-level VM parameters complicates the management of workloads within a datacenter. Rather, segmenting VM types allows the cloud provider to optimize the placement of VMs on individual machines within the datacenter. In particular, we assume that VM utilizations are dimensioned to be easy to partition (*e.g.*, a provider can choose to allow VMs with utilizations from one of 10%, 25%, 50%, or 100%). This means that bin packing is simplified and partitioning always succeeds. For example, an entire rack can be dedicated to 25% VMs of a particular client, with spare capacity for more, allowing for easy partitioning while enabling new VMs to be created.

For the sake of completeness, in the next two sections, we explain how Tableau deals with situations where partitioning does not succeed. The first of those techniques attempted when partitioning fails is *semi-partitioning* [9].

5.3 Semi-Partitioning

Semi-partitioning is a simple extension of partitioning. First, we try to partition the task set as before. However, when encountering a task that cannot be assigned to any core, instead of giving up, the task is broken up into smaller *subtasks* with precedence constraints, which are then easier to partition. The subtasks represent the task's fractional allocations on different cores. At runtime, a split task migrates among the cores to which its subtasks have been assigned to use the reserved processor time.

The trick is to ensure **(i)** that the subtasks never execute in parallel (since they still reflect the same sequential task), and **(ii)** that no core becomes overloaded. In general, this is not trivial, but many suitable semi-partitioning schemes have been proposed in recent years [10, 11, 16, 17, 20, 59, 60, 68].

Task	C	T	D	on core
τ_1	18	30	30	1
τ_2^a	12	60	12	1
τ_2^b	24	60	48	2
τ_3	72	120	120	2

TABLE 5.3: Semi-partitioned example tasks

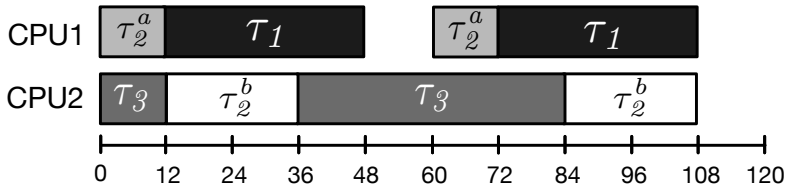


FIGURE 5.1: Example scheduling table for the semi-partitioned task set given in Table 5.3

We simply apply $C=D$ *task-splitting* [20] (see Chapter 2) that virtually always finds a valid split, even for difficult problem instances that almost fully utilize all cores [18]. Finding valid $C = D$ task splits is non-trivial, coNP-hard [41], and computationally demanding in general; however, due to the fixed table length, it is fast in Tableau's use case.

Running example. Recall that the task set in our example could not be partitioned onto two cores. Let us resolve the situation by selecting τ_2 with a cost of $36ms$ to be semi-partitioned. Applying the $C=D$ scheme, we manage to split the task into two subtasks τ_2^a and τ_2^b with costs of $12ms$ and $24ms$, respectively. Both subtasks can be assigned without causing overload and we can obtain a suitable table by simulating an EDF schedule on each core until the hyperperiod is reached (which in this example is simply 120). The resulting scheduling table is shown in Figure 5.1.

If semi-partitioning succeeds, the planner again simulates an EDF schedule on each core (including both tasks and subtasks), tracking the scheduling intervals and the vCPU associated with them. On the other hand if semi-partitioning fails, which only occurs for pathological configurations that are never seen in practice, we continue with *localized optimal scheduling*.

5.4 Localized Optimal Scheduling

While the $C = D$ approach is empirically near-optimal [18], *i.e.*, it is virtually always possible to find workable task splits, there nonetheless exists theoretically a chance that it might fail. In such a case, which we never encountered in our evaluation, it is possible to fall back to *optimal* multiprocessor real-time scheduling as a last resort [14, 50, 83, 97, 102]. Although optimal schedulers guarantee the existence of a schedule, we do not use them as our first choice since they tend to generate many preemptions and migrations. Instead, we perform semi-partitioning to the extent possible, and use an optimal scheduler to schedule the remaining tasks on a minimal subset of cores.

Specifically, we identify two physical cores that are "close" (*e.g.*, that share a cache) and turn them into a *cluster* (*i.e.*, a "double-sized bin") that is optimally scheduled. This merging of bins is repeated if needed until all tasks can be partitioned, split into sub-tasks, or assigned to some cluster of cores. The process is guaranteed to stop when reaching a single cluster encompassing all cores (if the system is not over-utilized). However, we emphasize that this procedure is virtually never needed for practical workloads; we include it simply so that table generation truly never fails (unless the system is over-utilized, which is a misconfiguration that is rejected).

It is worth mentioning that vCPUs that migrate among two or more pCPUs due to semi-partitioning (or localized optimal scheduling) represent a complication for the second-level scheduler—on which pCPU should such a vCPU participate in the second-level scheduling? To avoid costly synchronization, one straightforward approach is to adopt a "trailing core" policy: migrating vCPUs participate in the second-level schedule (only) on the pCPU on which they last received a guaranteed allocation. This also necessarily means that the vCPU would be assigned budget by the second-level scheduler for the particular core on which it is dispatched, and migrating across cores would require recalculating the vCPUs budget (only possible in the next epoch of the new core) to maintain fair-share properties.

5.5 Table Generation

The final step of the planner is to generate the actual scheduling tables. This is done in two steps: first a series of post-processing steps ensure that the slots in the table are within practical system constraints, and second the planner writes it to disk.

5.5.1 Post-Processing Tables

After a schedule has been found, the planner performs certain post-processing operations before handing the schedule over to the dispatcher. First, it coalesces allocations below a certain threshold into a neighboring allocation. This threshold is determined by the overheads involved in context-switching vCPUs, since allocations smaller than the threshold are impossible to enforce. In the last step, the planner "slices" the table to enable constant-time lookups, as discussed in the following section.

Finally, while this thesis does not explore this space, it is trivial to add additional post-processing steps to the current planner implementation. For instance, one might add a "peep-hole" optimization pass to reduce the number of migrations and preemptions even further. Alternatively, one might add a pass to encourage or discourage co-scheduling of certain VMs, *e.g.*, due to performance-counter-based profiles or for synchronization purposes. We leave these interesting opportunities and extensions to future work.

5.5.2 Storing and Pushing Tables

Recall that during each of the above phases, the planner keeps track of the simulated scheduling intervals and the vCPUs associated with each. Once post-processing of tables is complete, the planner simply writes these intervals to disk.

Rather than writing scheduling tables in a format that requires the hypervisor to parse and create an in-memory version of it, tables are written to disk as binary files mirroring the exact memory layout of data structures internally used by the hypervisor. Therefore, when the table is pushed to the hypervisor, and copied from dom0 memory into the hypervisor memory, no heavy processing needs to be performed to make it usable. The only processing that is performed is to walk through the list of vCPU structures in the table memory and rewrite placeholder pointers with the addresses of internal vCPU structure used by Tableau. A Python-based program that generates the memory layout for each table generated by the planner is shown in Appendix D.

5.6 Summary

In this chapter, we looked at how the planner in Tableau generates scheduling tables for use by our Xen-based dispatcher. The planner starts by modeling each vCPU as an equivalent periodic task, ensuring that the hyperperiod of the set of all vCPUs in the system is bounded. It then attempts to partition, and semi-partition them onto

individual cores, with localized optimal scheduling being a rare final case. Finally, it writes tables to disk as binary files mirroring a memory layout that requires minimal processing by the hypervisor to use.

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter, we present results from our experimental evaluation of Tableau. The presented results serve to validate the following key claims.

1. The time and space overheads of Tableau's planning step are acceptable relative to typical VM commissioning and decommissioning times.
2. Tableau incurs low scheduling overheads compared to other Xen schedulers.
3. Tableau offers both predictability (*i.e.*, consistent, low latencies) and high throughput in a high-density scenario compared to other Xen schedulers.
4. Tableau provides comparable or higher throughput for VMs compared to existing schedulers when configured with dedicated cores for each VM.
5. Tableau can be configured to provide performance guarantees for tier-2 background VMs.
6. Tier-2 VMs have a low impact on the performance of tier-1 VMs for the evaluated workload.

We begin with the time and space overheads of Tableau's planner.

6.1 Table-Generation Overheads

The time and memory overhead of Tableau's planner varies depending on **(i)** the number of VMs, and **(ii)** the configuration of individual VMs, and **(iii)** the number of cores in the system. Together, these parameters determine the number of slots and slices that need to be generated, optimized, and written to disk.

Rather than exhaustively evaluate the planner overheads under every potential configuration, we instead chose to demonstrate the *performance trends* it exhibits. In particular, we evaluated how **(i)** the number of VMs in the system and **(ii)** the choice of latency goals affected the time and space overheads of the planner. We do not evaluate the performance as a function of cores but instead present performance results from a machine with a large number of cores.

Specifically, we measured both the time taken to generate tables, as well as the size of the generated tables for a varying number of VMs, with all VMs being assigned one of four latency goals (1ms, 30ms, 60ms, and 100ms). To stress the planner and test its scalability limits, we performed these experiments on and for a 48-core Intel Xeon (E7-8857) server, the largest machine in our lab at the time of writing. Four cores were dedicated to dom0, and four tier-1 VMs were admitted for each of the remaining forty-four cores.

Table-generation time. In Figure 6.1, the Y axis shows the total time taken to generate the table (averaged over 100 separate runs) as a function of the number of VMs for which the table was generated. The number of VMs was varied up to a total of 176 VMs (*i.e.*, four VMs per core).

As can be seen in the figure, table-generation time never exceeds two seconds for the machine used in our evaluation. We believe this to be acceptable in the context of public clouds where typical VM lifetimes far outweigh VM startup, teardown, and reconfiguration times [29].

In scenarios where system re-configuration time may be crucial (*e.g.*, Tableau-based container scheduling, or even high-priority process scheduling), several optimizations could be made, both at an *implementation level* and the *design level*.

Implementation level. These techniques involve implementation-level optimizations that improve table generation times. This includes, **(i)** incrementally re-computing tables on a per-core basis, and **(ii)** reducing language runtime overhead (recall that the planner presented in this thesis was implemented in Python) by switching the implementation of the planner to a more performance-oriented compiled language such as C.

Design level. These techniques involve changing the design of the planner itself. First, as in Tableau the planner does not necessarily have to reside on the same machine (*i.e.*, table generation may also be offloaded to a faster, independent machine, similarly to how jobs are scheduled across data centers [99]), it is trivially possible to centrally cache tables for common configurations that are frequently reused. Second, for machines that run homogeneous VM types, or ones whose configuration is known beforehand, the planner can be extended to enable pre-creating slots for future VMs, which initially

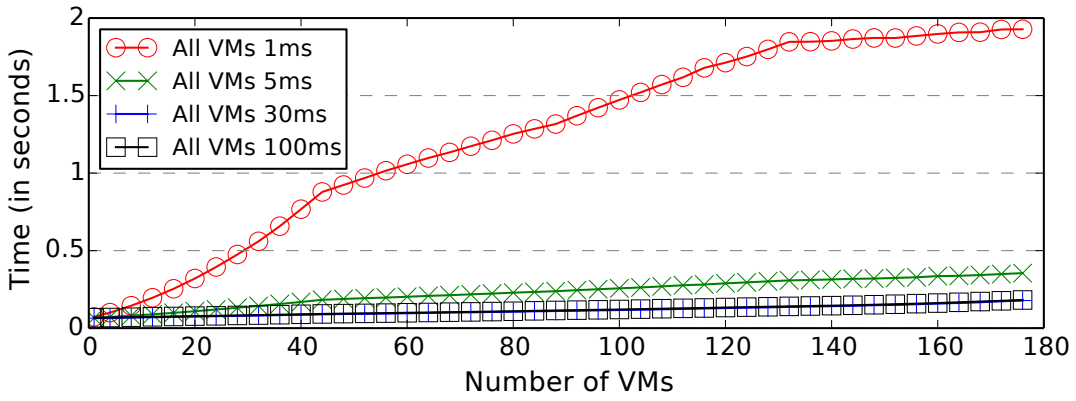


FIGURE 6.1: Table-generation times for a varying number of VMs with different latency goals. The 30 ms and 100 ms curves overlap.

point to the idle vCPU, and can be switched to a newly created VM with low overhead. This would render the overhead of table generation a one-time cost at system initialization, with tier-1 guarantees being near-instantaneous upon VM creation.

Increased reconfiguration delays. Since the planner resides locally in dom0, where it is triggered on demand when VMs are created, torn down or reconfigured, Tableau introduces a planning delay to these operations. However, we emphasize that the planning overhead does not affect the performance of VMs once they have commenced execution (*i.e.*, it increases only their provisioning or reconfiguration time). As VM creation under Xen already takes many seconds, even without accounting for the time it takes the guest OS to actually boot up (nor any time spend on fetching a VM image from remote storage), we deem even the longest table-generation delay reported in Figure 6.1, which is two seconds, to still be acceptable.

Memory overheads. Figure 6.2 shows the table size (in MiB) on the Y axis, as a function of the number of VMs on the X axis. The four curves show the table size when all VMs are all assigned a latency goal of 1ms, 30ms, 60ms, and 100ms, respectively.

As can be seen in the figure, the memory overhead for all configurations was below 1.2 MiB, which only occurs for a fairly demanding case of every VM having a latency goal of 1 ms. We consider this to be a negligible overhead for modern server-class machines with hundreds of gigabytes, and even terabytes, of RAM. We consider a latency goal of 1 ms to be unsuitable for application performance as it results in microsecond-scale slot sizes for VMs in the scheduling table. This results in a large number of scheduler activations at runtime, and as a result degrades performance. We show this empirically in Appendix A, where the performance of VMs under Tableau with a

1 ms scheduling latency result in the lowest performance compared to those measured across a wide range of more practical values.

Apart from low-level optimizations of reducing data structure sizes, this overhead is to some extent non-negotiable in that it is the least amount of information required at runtime to ensure O(1) scheduling in Tableau. It should be pointed out that this, in general, makes Tableau an unsuitable fit for embedded devices with severely limited memory (*e.g.*, 8-bit microcontrollers with kilobytes of RAM). However, as memory usage depends on the number of VMs, the number of cores in the system, and the specific configuration parameters for each VM, there may be situations where Tableau is acceptable for embedded use (*e.g.*, for a small number of VMs running on few cores with larger configuration parameters values). Further, Tableau's design would be suitable for many modern 32-bit embedded devices (*e.g.*, Raspberry Pi), which can have many gigabytes of memory.

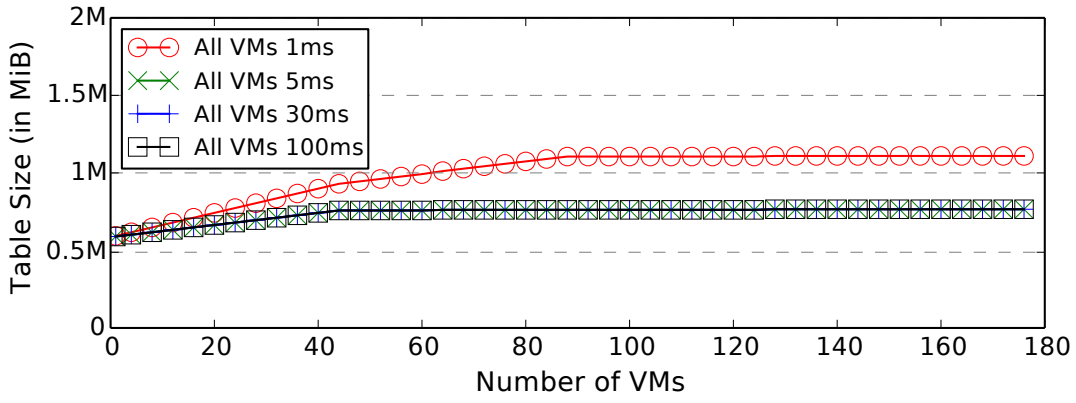


FIGURE 6.2: Generated table size for a varying number of VMs with different latency goals. All but the 1 ms curve overlap.

6.2 Scheduler Runtime Overheads

We now look at the runtime overheads of Tableau compared to other Xen schedulers, which is a crucial factor in the performance of VMs running on top of a hypervisor. This is because any cycles spent in the scheduler runtime are cycles that could have been used by some VM to perform useful work. While eliminating *all* runtime overhead is not possible (*e.g.*, interrupts still need to be handled and routed to VMs), from a scheduler standpoint, any cycles spent beyond guaranteeing the SLA of VMs are wasted cycles. Second, even when restricting the cycles spent within the scheduler to those operations needed to guarantee an SLA, the performance must be optimized to minimize the total number of cycles spent for this purpose.

To gain a better understanding of how Tableau performs, we present microbenchmarks comparing Tableau's scheduler overheads with three different schedulers in Xen.

Platform. We used a 16-core, 3.2 GHz Intel Xeon (E5-2667) server (comprising two sockets with eight cores each) with 512 GiB of RAM, running Ubuntu 16.04.3 LTS (Linux kernel version 4.4.0) on Xen 4.9. We employed an identical client machine, connected on the same network via 10 Gbit/s Ethernet, as a load generator. We disabled all CPU power-saving features for our evaluation to avoid performance unpredictability.

Evaluated schedulers. We compared our implementation of Tableau with three stock schedulers in Xen (Credit, Credit2, and RTDS). Recall that Credit is the default scheduler in Xen and is a weighted proportionate-fair-share scheduler. That is, each VM is allocated credits proportional to a configured weight, which it "burns" when it executes. Additionally, Credit gives VMs that wake up from an I/O operation a "boost" in priority. Recall that Xen's more recent Credit2 scheduler extends the original Credit design with the goal of improving responsiveness, and does this primarily by eliminating Credit's priority boosting as it is now understood to cause performance unpredictability. Finally, recall that RTDS is a real-time scheduler that, like Tableau, is also based on the periodic task model [72]. However, in contrast to Tableau, and similar to Credit, RTDS is a dynamic scheduler (*i.e.*, it makes all decisions online) based on an EDF policy. RTDS is an interesting baseline to compare against because it provides similar capabilities in terms of predictable control over latency and utilization, while representing an entirely different set of trade-offs due to its dynamic nature.

Scheduler setup. Due to the number of tunable parameters in each of the evaluated schedulers, and the resulting vast configuration space, we did not attempt to exhaustively evaluate every possible parameter combination. Rather, our evaluation is based on a single setup that is intended to be representative of the kind of workloads Tableau is designed to support.

Specifically, on our 16-core server, we assigned four single-vCPU VMs per core (*i.e.*, each with 25% CPU utilization), with four cores dedicated to dom0.

In the results shown in this section, Credit was configured according to documented best practices. In particular, we used a global timeslice of 5 ms under Credit as the default 30 ms value is known to be non-ideal for I/O workloads [25]. Under Tableau, to allow for a reasonably fair comparison with Credit, we chose a maximum scheduling latency of 20 ms since Credit with a 5 ms timeslice will, in the presence of four VMs per core, replenish all credits roughly once every 20 ms. This results in the planner picking

a period of roughly 13 ms with a budget of about 3.2 ms. To enable a direct comparison, RTDS was configured to match the parameters of Tableau.

Due to differences in capabilities of the various schedulers, we evaluated two distinct scenarios: a "capped" scenario, where VMs are configured with CPU-usage upper bounds (supported by Credit, RTDS, and Tableau), and an "uncapped" scenario, where a VM's CPU usage is not bounded (supported by Credit, Credit2, and Tableau). We used Ubuntu 16.04.3 LTS as the guest OS.

Overhead results. Under each scheduler, we traced the runtime cost of key scheduling operations in an I/O-intensive scenario, where each VM ran an I/O-intensive workload based on the well-known `stress` benchmark [111] for a duration of 60 seconds. Overhead samples were collected using Xen's built-in tracing framework by adding tracepoints around key operations within the scheduler.

Table 6.1 shows the mean overhead (in μs) of three scheduler operations on our 16-core server: **(i)** the time taken to make a scheduling decision, **(ii)** the time taken to process wake-up interrupts, and **(iii)** the time taken to perform any operations after de-scheduling a vCPU, such as sending re-schedule IPIs to another core.

TABLE 6.1: Average runtime overheads (in μs) for three key scheduler-related operations on a 16-core, 2-socket server.

	Credit	**Credit2**	**RTDS**	**Tableau**
Schedule	8.08	3.51	2.86	1.43
Wakeup	2.12	5.19	3.90	1.06
Migrate	0.32	5.55	9.42	0.43

As can be seen from the results, our focus on runtime efficiency in Tableau's design (Section 3.2) and the optimized, core-local implementation of Tableau's dispatcher (Chapter 4) is clearly reflected in its low scheduler overheads. We observe that Tableau indeed incurs substantially lower overheads compared to other schedulers: the mean scheduling overhead under Tableau is around 5.6x, 2.4x, and 2x lower than under Credit, Credit2, and RTDS, respectively. Concerning post-scheduling operations ("Migrate"), recall that Tableau may occasionally need to send an IPI after de-scheduling a vCPU. As expected, this results in only a negligible increase in the overhead (approximately an additional 100ns on average compared to Credit in our example).

RTDS incurs significantly higher overhead (over 9μs) for post-schedule work due to requiring the acquisition of a global lock when load-balancing vCPUs. To highlight this bottleneck, we also collected overhead data on a 48-core server machine with four sockets (each comprised of 12 cores). Table 6.2 shows the observed overheads. It is obvious that RTDS' global lock does not scale well: on average, RTDS spends *over*

168µs while attempting to migrate a VM each time it is preempted. We do not present results from this machine any further in the remainder of this section.

TABLE 6.2: Average runtime overheads (in µs) for three key scheduler-related operations on a 48-core, 4-socket server.

	Credit	Credit2	RTDS	Tableau
Schedule	16.40	4.70	4.39	2.49
Wakeup	7.07	5.61	19.16	1.82
Migrate	0.42	18.19	168.62	0.66

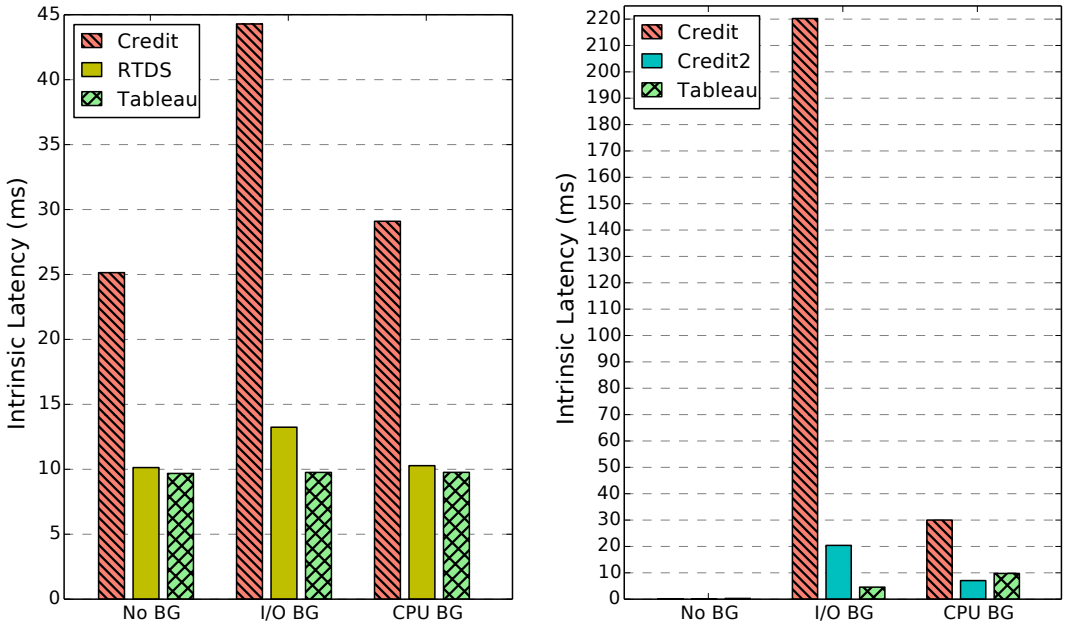
Finally, Table 6.1 shows the mean overhead for processing wakeups to be 2× lower compared to Credit, almost 5× lower compared to Credit2, and over 3× lower than when running RTDS. This is a consequence of Tableau's fast wakeup handling (Chapter 4), which uses the table to determine which CPU to send an IPI to.

To summarize, the advantages of Tableau's design choices are reflected in its efficient runtime compared to other schedulers. This is the result of moving VM budget and latency enforcement to an offline planner, using per-core data structures, and the use of a minimal table-driven dispatcher.

6.3 Comparing Scheduling Delay

To understand the scheduling delays induced by the existing Xen schedulers and Tableau, we used **(i)** the popular `redis-cli` workload with the `-intrinsic-latency` option, and **(ii)** measured the ping latency between our client machine and one of the VMs. The two workloads were chosen as they allow insight into the performance of respectively CPU-bound and sporadically activated, network-I/O-centric VMs under each scheduler. In the following, we present measurements from a single *vantage VM*. The vantage VM did not receive any special treatment or configuration-specific advantages and is thus representative of general scheduler performance.

`redis-cli` intrinsic latency. `redis-cli` is a command-line interface distributed as part of the `redis` key-value store. We ran it within our vantage VM and measured the *intrinsic latency* of the system. When measuring the intrinsic latency, `redis-cli` runs a tight CPU-bound loop and measures the delay between iterations, thus measuring if any delays occur due to the scheduler. That is, if the scheduler preempts the VM during a particular loop, the end time will only be recorded once the VM has been dispatched again. This allows us to precisely measure the scheduling delay that each scheduler induces.



(A) Capped VMs

(B) Uncapped VMs

FIGURE 6.3: Maximum scheduling delay as measured by `redis-cli`. "BG" denotes background workload.

To isolate the effect of the VM scheduler, we ran the tool with the highest SCHED_FIFO priority to avoid interference arising from the Linux scheduler in the guest VM. We evaluated both capped and uncapped scenarios, with four VMs per core, without any background workload, with an I/O-intensive background workload, and with a CPU-intensive background workload. The background workloads used the popular `stress-ng` tool to spawn a single worker performing either I/O-intensive (using `-io 1`) or CPU-intensive work (using `-cpu 1`). The results are illustrated across two graphs, Figure 6.3a and Figure 6.3b, as Credit2 does not support capped VMs and RTDS does not support uncapped VMs.

In the capped scenario shown in Figure 6.3a, regardless of the background workload, every scheduler induces scheduling delays as it forcibly cuts off CPU access to VMs once they exceed their assigned amount. In the case of Credit, the VM experiences delays of up to almost 44 ms. Under RTDS, configured as discussed in Section 6.2, this results in around 10 ms in the best case with no background workload (*i.e.*, the VM runs at the beginning of each period); more latency (up to 13ms) was observed in the presence of a background workload. Finally, under Tableau, we always see about 10 ms of scheduling delay, regardless of background workload. In this experiment, RTDS controls scheduling latency just as well as Tableau, but we will later show that it does not achieve the same throughput.

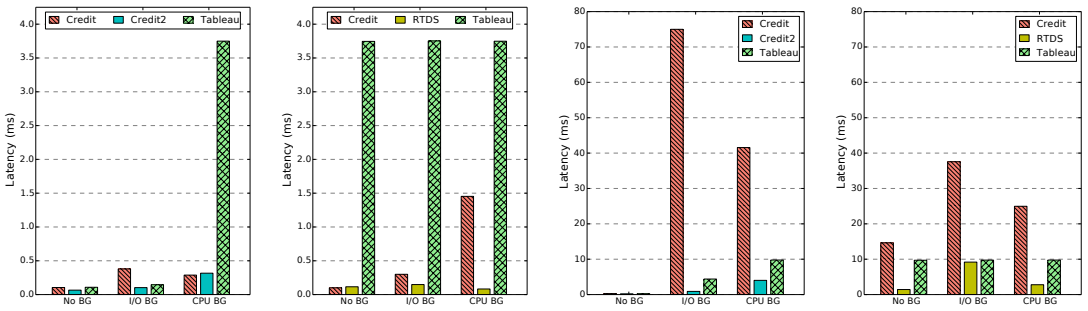
In the uncapped scenario shown in Figure 6.3b, VMs are not rate-limited and are allowed to consume additional idle cycles if available. As a result, when no background workload is present, all schedulers achieve sub-millisecond scheduling latencies, and the corresponding bars are barely visible in Figure 6.3b. However, latency becomes substantially worse under Credit and Credit2 as a background workload is introduced. In this case, the responsibility of maintaining low scheduling latency for all VMs falls on the scheduler, and as can be seen, it does not work well in high-density scenarios: we observed delays as high as 220 ms under Credit. Credit2 fares well in the presence of a CPU-intensive background workload, but not so well in the presence of the I/O-intensive workload. In contrast, under Tableau, the burden of meeting scheduling latency bounds is the responsibility of the semi-offline planner, which is oblivious to background workloads. As a result, Tableau exhibits at most 10 ms of scheduling delay regardless of background workload.

Ping latency. To cross-validate our findings, we also measured the average and maximum observed ping latency from our client machine to the vantage VM. ICMP echo requests are handled directly within the guest kernel, which eliminates any dependence on the guest scheduler, but can only be processed when the VM is dispatched by the VM scheduler. As a result, with a controlled network like in our setup, the ping latency is dominated by (and is a good proxy for) the scheduling latency incurred by a VM in reaction to wake-ups triggered by external I/O events.

We again evaluated both capped and uncapped scenarios, without any background workload, with an I/O-intensive background workload, and with a CPU-intensive background workload. Similar to the previous experiment, the background workloads again used the popular `stress-ng` tool to spawn a single worker performing either I/O-intensive (using `-io 1`) or CPU-intensive work (using `-cpu 1`).

The experimental setup consisted of eight threads on our client machine, each sending 5,000 randomly-spaced pings with delays ranging from zero to 200 ms. The resulting 40,000 samples were aggregated to determine the average and the maximum observed ping latency for each configuration. The results are reported in Figure 6.4a through Figure 6.4d.

In the uncapped scenario, without a background workload, the average latency (Figure 6.4a) is low for all schedulers (around 100 μs) as the VM can always react immediately to incoming packets. In contrast, the capped scenario (Figure 6.4b) shows the impact of the table's rigid structure, which results in Tableau exhibiting clearly higher average latency (but well below the configured latency goal of 20 ms).



(A) Uncapped VMs, Average

(B) Capped VMs, Average

(C) Uncapped VMs, Maximum

(D) Capped VMs, Maximum

FIGURE 6.4: Average and maximum-observed round-trip ping latencies. "BG" denotes background workload.

With an I/O workload in the uncapped scenario, since background VMs frequently block, the vantage VM is able to leverage the resulting idle cycles to achieve a low average latency (Figure 6.4a). In the case of a CPU-bound background workload, however, there are no additional idle cycles to be had and the vantage VM can only execute during its own slots, which are active only periodically. Thus, the average latency under Tableau is noticeably higher (Figure 6.4a), but still well below the configured latency goal of 20 ms), since the average latency is determined by the gaps between slots in the table. In contrast, under the other schedulers, which are dynamic in nature and employ heuristics that favor I/O workloads (as is the case under Credit), the vantage VM is able to (on average) respond almost immediately since it is allowed to preempt the predominantly CPU-bound background VMs. However, as we show in Section 6.4, these same features can also reduce application throughput and lead to increased unpredictability.

In the uncapped scenario, the maximum observed latencies in an otherwise idle system are around 200 μs (Figure 6.4c). However, once a background workload is introduced, the maximum observed latency increases under all schedulers. Under Credit, we observe latencies approaching 75 ms in the presence of an I/O-intensive background workload (Figure 6.4c). Credit2 continues to provide good tail latency characteristics, but as we will show in Section 6.4, it is unable to maintain high throughput in this scenario.

In the capped scenario (Figure 6.4d), the maximum observed latencies under Credit are significantly higher even without any background workload. This is simply because, while VMs are not running any benchmark, they still require CPU time occasionally for system processes. As a result the vantage VM may, under rare circumstances, exhaust its budget, while simultaneously having to wait for the other three background VMs on the same core to exhaust their budget, resulting in up to 15 ms of scheduling latency.

While in principle RTDS is also susceptible to the same worst-case behavior, the necessary conditions did not trigger during our experiment because they occur only very rarely.

With an I/O background workload active, Credit exhibits tail latencies of around 30 ms (Figure 6.4d). On the other hand, RTDS and Tableau enable accurate control over the scheduling delay. The maximum observed ping latency under RTDS is around 9 ms (Figure 6.4d), somewhat less than the delay allowed in each period. Similarly, regardless of the background workload, Tableau never exhibits latencies above 10 ms (Figure 6.4d), which reflects the structure of the table that the planner created for this workload.

To summarize, Credit shows substantially increased tail latency under load in a high-density scenario. While Credit2 and RTDS show good latency characteristics, they struggle to do so while maintaining high throughput, as we show next.

6.4 Comparing `nginx` HTTPS Throughput (High Density)

We now present a comparison of Tableau, RTDS, Credit, and Credit2 in a high-density scenario in terms of their respective impact on *application* throughput and latency, as exemplified by the `nginx` web server.

We used `wrk2` [2], an extension of the well-known `wrk` HTTP load generation tool, that allows for accurate measurement of tail latencies while accounting for the *Coordinated Omission* problem [105].

Setup. Our setup again comprised of four single-core tier-1 VMs per core on twelve cores of our two-socket, 16-core server (*i.e.*, a total of 48 VMs), with the remaining four cores being dedicated to dom0. Each VM was assigned a virtual network interface using Intel's SR-IOV technology that allowed it to bypass the I/O scheduler in dom0. There were no tier-2 background VMs present, which we evaluate separately in section 6.6.

The vantage VM was hosting an `nginx` server that served a small PHP "application" via HTTPS. The PHP application simply sends a randomly selected file of a given size (1 KiB, 100 KiB, or 1 MiB) chosen from a 1 GiB dataset. To minimize measurement noise, all files were stored in an in-memory `tmpfs` volume. Similarly, `nginx` was assigned a real-time priority to reduce noise in the results due to activity of the guest OS's scheduler.

The client machine hosted the `wrk2` tool, which generated requests for a specific file size (1 KiB, 100 KiB, or 1 MiB) at a given rate, and measured the achieved throughput and latency characteristics of the requests. We increased the request rate progressively until the server was saturated. As in the previous experiments, we evaluated both capped and uncapped scenarios, with and without an I/O-intensive background workload, which was generated using the popular `stress-ng` tool with a single I/O-intensive thread (via the `-io 1` command-line argument).

Note. To validate that the scheduler configurations used in this evaluation did not penalize any specific scheduler, and to validate that the benefits of Tableau are not limited to the specific configuration used in this section, a more exhaustive evaluation was also conducted. We compared the performance of Credit, Credit2, RTDS, and Tableau across various scheduling latency values (1, 5, 10, 15, 20, 25, 30, and 35 milliseconds), with different background workloads (idle, cache-intensive CPU-bound, and I/O-intensive) under the high-density setup described above. The results of this exhaustive evaluation can be found in Appendices A and B, and they support the findings presented in this chapter.

Graphs. The results are illustrated in Figure 6.5, comprising three columns and six rows. The first three rows (Figure 6.5(a)–(i)) show to the capped scenario and the last three rows (Figure 6.5(j)–(r)) show the uncapped scenario. Each row corresponds to the results for either 1 KiB files, 100 KiB files, or 1 MiB files. Within each row, the three columns correspond to the mean, 99$^{\text{th}}$ percentile, and the maximum observed latency, respectively, versus the observed throughput.

Since Credit2 does not support capped VMs and RTDS does not support uncapped VMs, each graph comprises only three curves: Credit, RTDS, and Tableau (for capped scenarios), and Credit, Credit2, and Tableau (for uncapped scenarios). The X-axis shows the observed throughput, while the Y-axis shows a latency metric. Thus, lower is better (*i.e.*, less latency), as is being further to the right (*i.e.*, higher throughput). At some point, the server can no longer keep up with the request rate, and the curve peaks upwards as queueing delays start to dominate.

Experiment 1: Capped VMs (Figure 6.5(a)–(i)). In the following, we discuss results for 1 KiB and 100 KiB files (the first two rows); we revisit Tableau's performance with 1 MiB files (Figure 6.5(g)–(i)) later in Section 6.7. We make the following key observations.

Tableau provides good tail latencies. The 99$^{\text{th}}$ percentile and the maximum observed latency under Tableau are lower than under Credit and RTDS (Figure 6.5(b)–(c) and

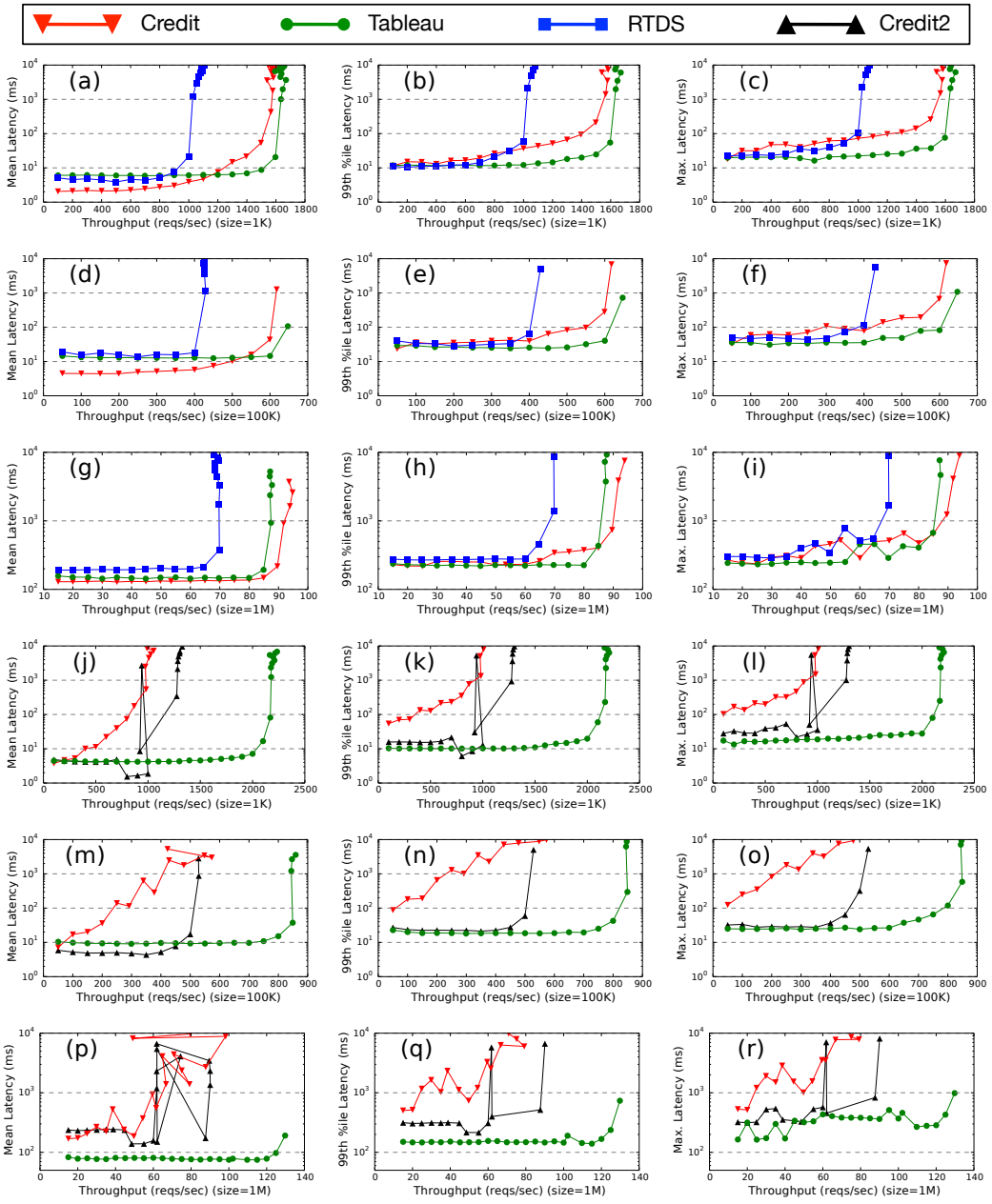


FIGURE 6.5: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for capped (first three rows) and uncapped scenarios (last three rows), for 1 KiB, 100 KiB, and 1 MiB files (see Y-axis labels), with an I/O-intensive background workload and with varying throughput.

Figure 6.5(e)–(f)). While for low request rates, Credit and RTDS's tail latencies are sometimes on par with Tableau's, they quickly increase with the request rate. In contrast, Tableau continues to maintain relatively stable tail-latency characteristics until the server reaches its peak throughput.

Tableau supports higher SLA-aware peak throughput. In both the 1 KiB and 100 KiB scenarios, Tableau achieves a higher peak throughput. In addition, Tableau's latency begins to creep upwards much later than under Credit and RTDS. Thus, given a latency-based

service-level agreement (SLA), Tableau supports a higher SLA-aware throughput. For example, for 1KiB files, given an SLA that mandates a 99th-percentile latency of 100ms or lower, the peak throughput for RTDS and Credit is around 1,000 and 1,400 requests per second, respectively (see Figure 6.5(b)). Tableau can support up to 1,600 requests per second while satisfying the SLA.

Tableau's rigidity affects its mean latency. The mean latency (Figure 6.5(a) and (d)) under Tableau is higher than under either Credit or RTDS for low request rates. This is expected in a table-driven scheduler, as a request arriving just after the end of a VM's slot has to wait until the next slot of the VM to be processed, while dynamic schedulers like Credit and RTDS can react to the request immediately. However, both schedulers become overwhelmed as the request rate increases, while Tableau's rigidity becomes advantageous and translates into stability at higher request rates.

RTDS struggles to sustain high throughput. In the presence of an intense I/O background workload that causes frequent scheduler invocations, RTDS achieves significantly lower peak throughput than either Credit or Tableau. This is apparent for all three file sizes, and highlights that high VM scheduling overheads can substantially reduce guest application performance.

Credit is significantly less predictable. Mean, 99^{th}, and maximum observed latencies under Credit start to increase significantly before peak throughput is reached. For instance, in graphs (b), (c), (e), and (f) of Figure 6.5, Credit exhibits a noticeable upwards slope before peaking (note the log scale), which reflects upon increasing unpredictability as the system becomes increasingly busy. This supports the observation that Credit's I/O boosting heuristic can backfire when faced with interference from I/O-intensive workloads.

Experiment 2: Uncapped VMs (Figure 6.5(j)–(r)). Recall that, in the uncapped scenario, Tableau allows uncapped tier-1 VMs to additionally execute in any idle time available on its core, with multiple contending VMs being allocated idle time in a round-robin manner. The challenge for the scheduler is thus to ensure that interference and overheads do not consume precious CPU cycles, thereby degrading the performance of the system. This is where Tableau's low-overhead, table-driven design shines: it maintains stable latency characteristics for significantly higher throughputs compared with Credit and Credit2 for all file sizes. We detail our observations below. Similar to Experiment 1, our setup for this experiment consists only of tier-1 VMs; tier-2 background VMs are evaluated separately in section 6.6.

Tableau supports significantly higher throughput. In all cases, Credit's performance starts to degrade already at a very low throughput. While Credit2 performs well at low

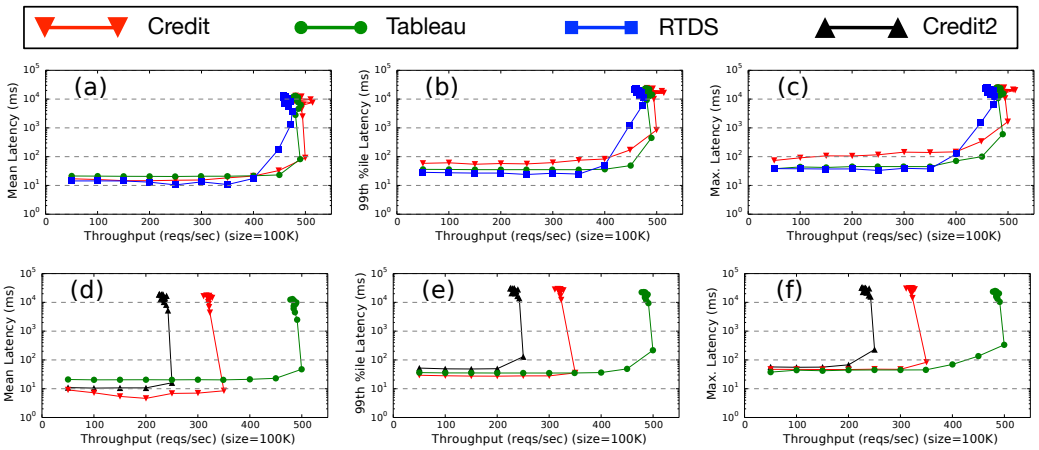


FIGURE 6.6: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for capped (first row) and uncapped VMs (second row) for 100 KiB files with a cache-thrashing background workload and varying throughput.

throughput, the peak throughput achieved under Credit2 is still considerably less than the peak throughput achieved under Tableau. For example, with 100 KiB files and a 99th-percentile SLA of 100 ms (Figure 6.5(n)), Credit supports only 50 requests per second, Credit2 supports up to 500 requests per second, but Tableau is able to support more than 800 requests per second, about 60% more than Credit2.

Tableau's second-level scheduler is effective. From Figure 6.5(n) and Figure 6.5(e), we can see that Tableau achieves a higher peak throughput of around 850 requests per second in an uncapped scenario compared with around 600 requests per second in the capped scenario. This is due to Tableau's second-level scheduler, which allows the vantage VM to use any idle cycles in the system in addition to its pre-determined slots. To evaluate the contributions of the second-level scheduler, we traced Tableau's scheduling decisions while fixing the request rate at 700 requests per second (supported by Tableau only in the uncapped scenario). We observed that over 85% of the scheduling decisions resulting in the vantage VM's execution were made by the level-2 round-robin scheduler. That is, idle cycles are efficiently and opportunistically allocated by Tableau to other VMs on the same core, which translates into improved throughput without a significant latency penalty.

Experiment 3: Cache-thrashing background workload. We now contrast how the schedulers perform in the presence of `stress-ng`'s cache-thrashing background workload, which is fully CPU-bound. Figure 6.6 shows the results for 100 KiB files; one row for the capped scenario (Figure 6.6(a)–(c)) and one row for the uncapped scenario (Figure 6.6(d)–(f)). Results for other file sizes were similar; we summarize the main trends.

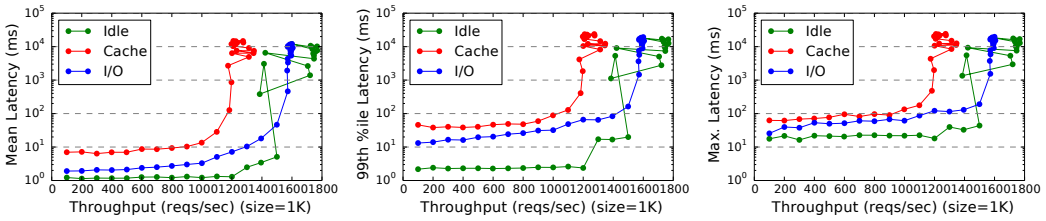
All schedulers perform similarly in the capped scenario. Since the background workload now is fully CPU-bound—none of the cache-thrashing background VMs ever voluntarily triggers the VM scheduler—the rate of scheduler invocations, and hence the impact of scheduling overheads, is much reduced. As a result RTDS fares much better, and can perform more or less as well as the other schedulers. Fundamentally, Figure 6.6(a)–(c) show a case where the VM scheduler is hardly a bottleneck, and hence it is not surprising to see little differentiation among the schedulers.

Credit outperforms Credit2 due to effective boosting in the uncapped scenario. Credit's boosting heuristic was ineffective in the prior experiment (Figure 6.5) since all VMs were I/O-bound and thus all (or effectively none) were prioritized. However, with a cache-thrashing background workload, Credit's boosting heuristic works as intended and plays in favor of the vantage VM, which is the sole VM performing I/O. On the other hand, Credit2, which does not explicitly favor I/O workloads, achieves a lower peak throughput, as can be seen in Figure 6.6(d)–Figure 6.6(f).

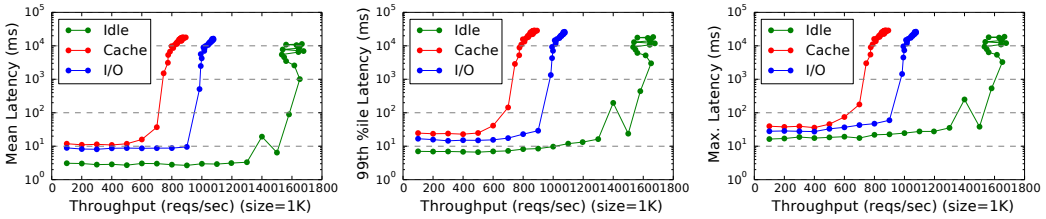
Finally, *Tableau outperforms both Credit and Credit2 in the uncapped scenario.* This is where Tableau's rigid table-driven design works best compared to Credit and Credit2's dynamic, heuristic-based designs, which struggle to maintain fairness given the aggressive CPU demand of the uncapped background workload. When comparing the peak throughput under Tableau in the capped and uncapped scenarios (first row vs. second row), we see no drop in Tableau's peak throughput (around 500 requests per second in both cases) as the vantage VM is guaranteed its utilization in both cases, while both Credit and Credit2 see a significant reduction in throughput due to increased interference from uncapped background VMs.

This experiment demonstrates Tableau's advantage in ensuring that each VM receives its guaranteed minimum amount of service no matter what the rest of the system is doing.

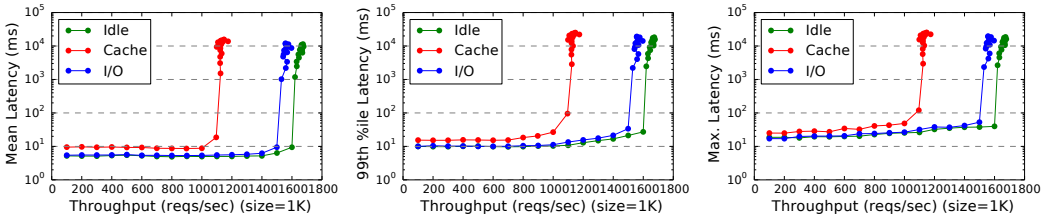
Experiment 4: Comparing susceptibility to interference. We now contrast the susceptibility of each scheduler to different types of interference from background VMs. Figures 6.7 and 6.8 show the results for 1 KiB files comparing the throughput-vs-latency curves under three different background workloads (idle, CPU-bound cache intensive, and I/O intensive). Figure 6.7 shows the results for Credit, RTDS, and Tableau in a capped scenario, while Figure 6.8 compares Credit, Credit2, and Tableau in an uncapped setting. Note that since an uncapped scenario with an idle background is similar to a dedicated-core scenario, and not comparable, we omit it from the graphs in Figure 6.8. We summarize the main trends below.



(A) Credit Scheduler



(B) RTDS Scheduler

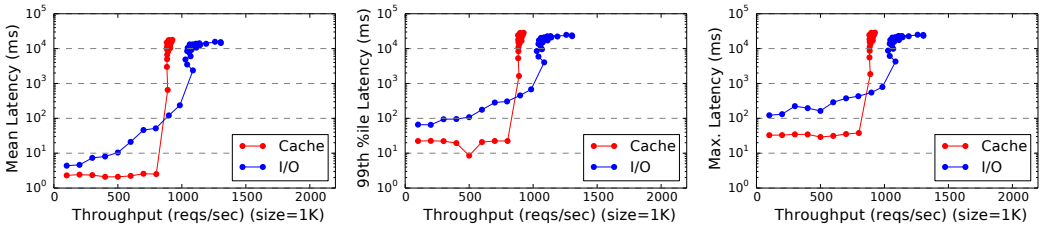


(C) Tableau Scheduler

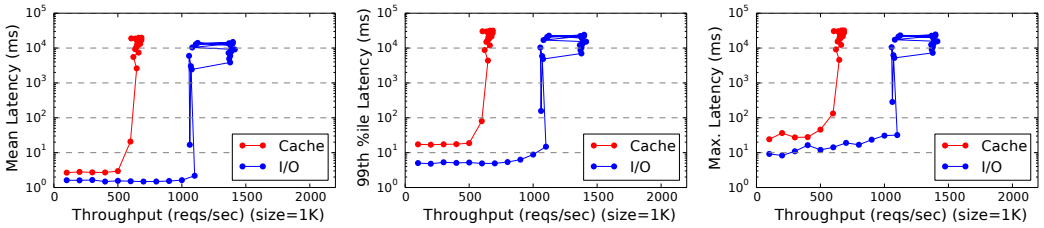
FIGURE 6.7: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped, high-density scenario for 1 KiB files, with three background workloads (I/O-intensive, CPU-bound cache-intensive, and idle) and with varying throughput.

Tableau provides better latency characteristics under interference in the capped scenario compared to Credit. Tableau's table-driven design, the latency characteristics are not dependent on scheduler decisions, but rather only on the background interference. In Figure 6.7(C), Tableau's curves are relatively flat until they rise sharply. In contrast Credit (Figure 6.7(A)) sees a gradual rise in latency in the presence of an cache-intensive or I/O-intensive background workload. Note that Credit provides comparable peak throughput as Tableau (around 1,500 requests per second), although it achieves this at the cost of increased tail latencies. In fact, as is evident from comparing Figure 6.7(A) and Figure 6.7(C), Tableau outperforms Credit on 99th percentile latency when under interference, and while mean latencies under Credit are lower for lower request rates, Tableau provides *more consistent* mean latencies across the entire throughput range.

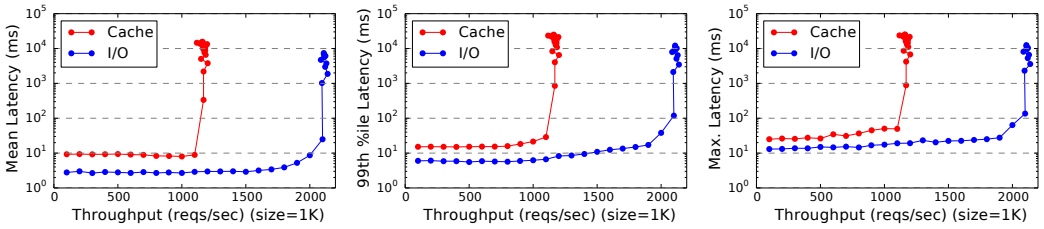
Tableau provides higher throughput under interference in the capped scenario compared to RTDS. When comparing Tableau (Figure 6.7(C)) with RTDS (Figure 6.7(B)), we see that Tableau supports significantly higher throughput under interference. For example, with an I/O background workload, the throughput under Tableau peaks at around



(A) Credit Scheduler



(B) Credit2 Scheduler



(C) Tableau Scheduler

FIGURE 6.8: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for an uncapped, high-density scenario for 1 KiB files, with two background workloads (I/O-intensive and CPU-bound cache-intensive) and with varying throughput. Results with the idle background workload are omitted as they are equivalent to a dedicated core scenario, and therefore not comparable.

1,500 requests per second compared with RTDS, which peaks at around 1,000 requests per second, almost a 30% reduction.

Tableau provides higher throughput under interference in the uncapped scenario compared to both Credit and Credit2. When comparing Tableau (Figure 6.8(C)) with Credit and Credit2 (Figure 6.7(A) and (B)), we see that Tableau supports significantly higher throughput under interference even in the uncapped scenario. For example, with an I/O background workload, the throughput under Tableau peaks at around 2,200 requests per second compared with Credit, which peaks at around 1,000 requests per second (around 50% lower) and Credit2, which peaks at around 1,100 requests per second (around 40% lower). Again, this is due to how uncapped VMs receive guaranteed service via the generated table as well as additional service when the currently-scheduled tier-1 VM is not runnable (*e.g.*, when it blocks on I/O).

All schedulers suffer under interference, particularly cache-intensive interference. As can be

seen in both Figure 6.7 and Figure 6.8, all four schedulers, including Tableau, see significant reduction in throughput, as well as a relative increase in latency when subject to cache interference from background VMs. However, it should be noted that Tableau performs significantly better than the other schedulers. In the capped scenario, Tableau's peak throughput is higher compared to RTDS, and while Credit matches it, it does so at the cost of increased latency. In the uncapped scenario, Tableau outperforms both Credit and Credit2.

In summary, while Tableau experiences throughput variations as a result of background interference, it performs comparably or better with regard to peak throughput relative to the other evaluated schedulers. Where it performs comparably on peak throughput, it outperforms on latency characteristics, ensuring lower and more predictable latencies.

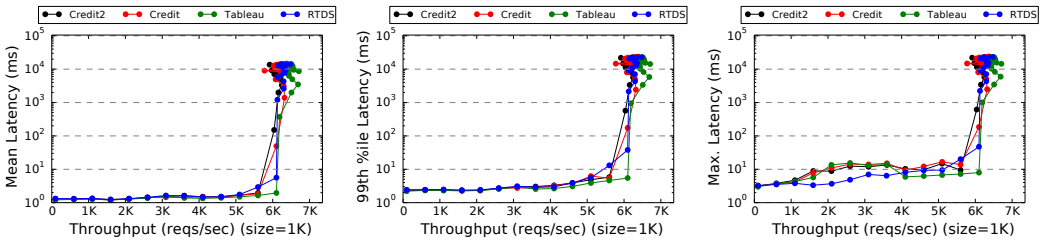
6.5 Comparing `nginx` HTTPS Throughput (Dedicated)

While the previous section shows the advantages of Tableau in a high-density scenario, we also evaluate its performance in the common-case scenario where VMs are assigned dedicated cores (*i.e.*, there is a one-to-one mapping between vCPUs and physical CPUs). Such dedicated-core scenarios are popular as they allow for simpler configuration of individual VMs. In particular, since there is only a single VM per core, scheduling latency arising due to the multiplexing of VMs is absent entirely, and does not need to be configured. In this section we evaluate Tableau in a dedicated-core scenario to determine whether its performance in high-density settings comes at the cost of performance degradation in the common-case, dedicated-core scenario.

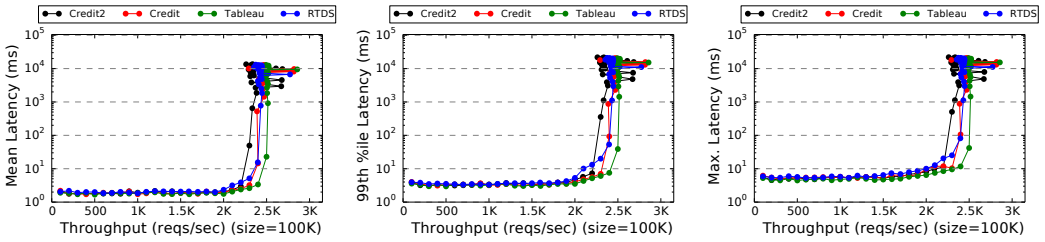
We present evaluation results from such a scenario, comparing the performance of Tableau against the three evaluated Xen schedulers, and show that Tableau's advantages extend beyond high-density scenarios and to dedicated-core ones as well.

Experimental setup. Our setup comprised of one single-core VM per core on twelve cores of our two-socket, 16-core server (*i.e.*, a total of 12 VMs), with the remaining four cores being dedicated to dom0. Each VM was assigned a virtual network interface as in the high-density experiment.

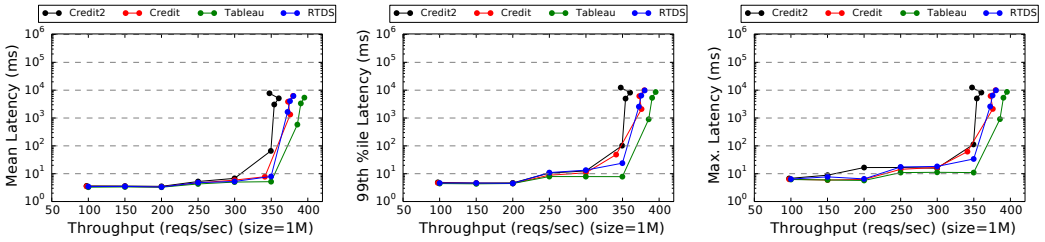
As in the high-density experiment, the vantage VM was hosting an `nginx` server that served a small PHP application via HTTPS, which responded with a randomly selected file of a given size (1 KiB, 100 KiB, or 1 MiB) chosen from a 1 GiB dataset. To minimize measurement noise, all files were stored in an in-memory `tmpfs` volume. Similarly,



(A) 1 KiB Files



(B) 100 KiB Files



(C) 1 MiB Files

FIGURE 6.9: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency with a dedicated-core scenario for 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying throughput.

`nginx` was assigned a real-time priority to reduce noise in the results due to activity of the guest OS's scheduler.

The client machine hosted the `wrk2` tool, which generated requests for a specific file size (1 KiB, 100 KiB, or 1 MiB) at a given rate, and measured the achieved throughput and latency characteristics of the requests. The request rate was increased progressively until the server was saturated. Unlike the previous experiments, since there is no difference in a dedicated-core setting between capped and uncapped scenarios, we configured VMs under Credit, Credit2, and Tableau to be uncapped, while they were "capped" at 100% under RTDS (as it does not support uncapped VMs). The remaining VMs ran an I/O-intensive background workload using the popular `stress-ng` tool.

Note. A more exhaustive evaluation was conducted to compare the performance of Credit, Credit2, RTDS, and Tableau with different background workloads (idle, cache-intensive CPU-bound, and I/O bound) under the above-described dedicated-core scenario. The results of this exhaustive evaluation can be found in Appendix C. We note

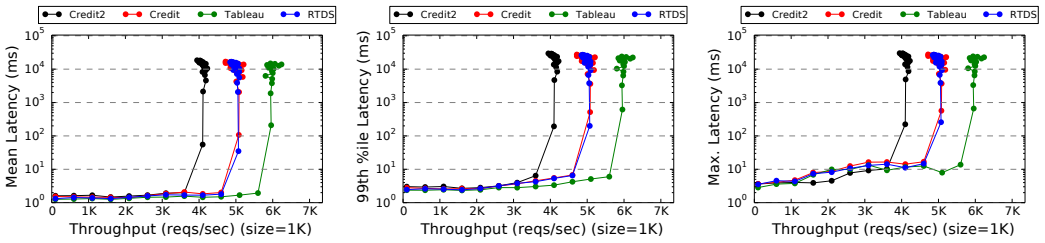
that the trends presented in this section are consistent with the results for the larger range of configurations evaluated in Appendix C.

Graphs. The results are illustrated in Figure 6.9, comprising three rows of graphs with three columns each. Each row corresponds to the results for different file sizes being requested (1 KiB files, 100 KiB files, or 1 MiB). Within each row, the three columns correspond to the mean, 99th percentile, and the maximum observed latency, respectively, versus the observed throughput. Each graph comprises four curves, one for each of the evaluated schedulers (Credit, RTDS, Credit2, and Tableau). As the X-axis shows the observed throughput, and the Y-axis a latency metric, lower is better (*i.e.*, less latency), as is being further to the right (*i.e.*, higher throughput).

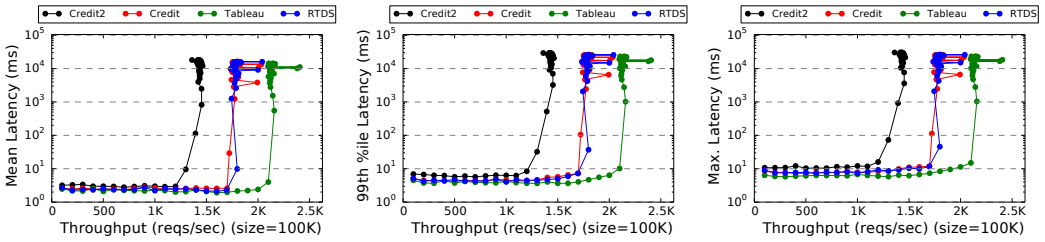
Tableau performs comparably or better than existing schedulers. As can be seen in the figures, in a dedicated-core scenario, Tableau performs comparably or better than the other evaluated schedulers in both throughput and latency metrics, regardless of the request size. In all figures, it can also be observed that the peak throughput under Tableau is slightly higher, and the mean and 99th percentile latencies are more predictable compared to the other schedulers. This is owed to the lower runtime overheads incurred by Tableau compared to other schedulers, combined with the simplicity of scheduling dedicated VMs; as no multiplexing of CPU time between multiple VMs is required, VMs do not experience any scheduling delays due to other VMs on the same core.

VMs under Tableau experience the lowest slowdown under cache interference compared to other schedulers. Figure 6.10 shows the results from a setup similar to the one used in Figure 6.9, with the only difference being that all other VMs in the system run a cache-intensive `stress-ng` workload instead of an I/O-intensive one. As can be seen from the graphs in Figure 6.10, Tableau outperforms all other schedulers on throughput while providing comparable or better latency guarantees. For example, for 100 KiB files, Tableau achieves around 2,100 requests per second, while Credit and RTDS peak at approximately 1,800 requests per second. Credit2 achieves the lowest peak throughput of around 1,500 requests per second. Contrasting these results with Figure 6.9 shows that Tableau results in the least performance degradation under cache-intensive background stress compared to an I/O one. Compared to 1 KiB files under an I/O-intensive workload, Tableau sees a reduction from around 6,500 requests per second to 6,000 requests per second (an 8% reduction). In contrast, Credit sees a reduction from 6,000 requests per second to just 5,000 requests per second (a 16% reduction).

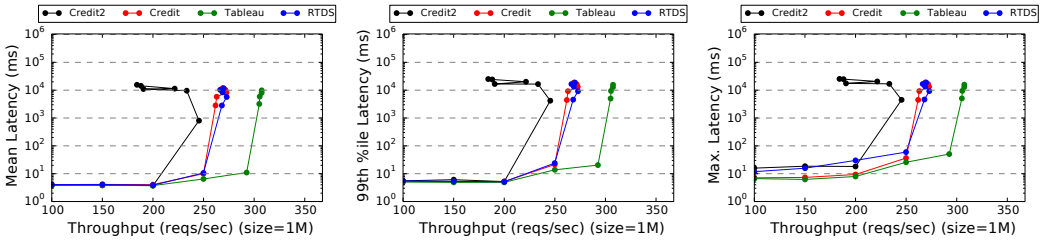
To summarize, Tableau does not sacrifice performance for common-case configurations in order to provide high performance for high-density configurations. Further, it provides additional benefits: slightly increased throughput and more predictable latencies



(A) 1 KiB Files



(B) 100 KiB Files



(C) 1 MiB Files

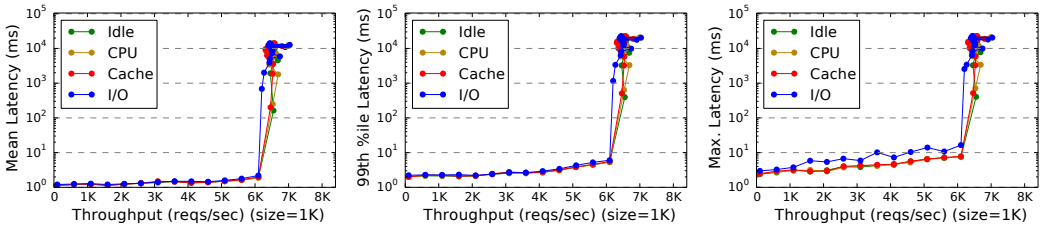
FIGURE 6.10: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency with a dedicated-core scenario for 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying throughput.

in many cases. Further, while all schedulers see throughput degradation when co-located with a cache-intensive background workload instead of an I/O-intensive one, Tableau achieves the least degradation.

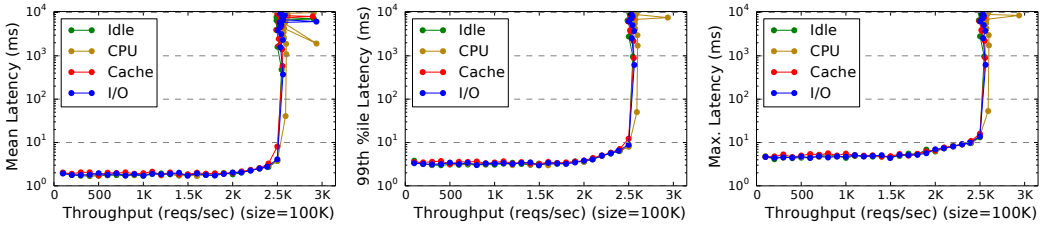
6.6 Tier-2 Performance and Impact

In this section, we evaluate both the performance of tier-2 VMs as well as their effect on the performance of (tier-1) table-driven VMs.

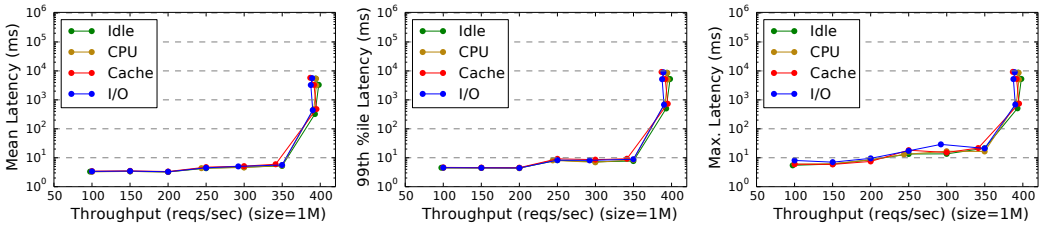
A primary goal of running background VMs in a datacenter to use up any available idle cycles is to avoid performance degradation of VMs that have guaranteed performance. In particular, since the most common configuration in commercial clouds is dedicated-core VMs, we evaluate the impact of tier-2 VMs on the performance of dedicated-core tier-1 VMs in this section.



(A) 1 KiB Files



(B) 100 KiB Files



(C) 1 MiB Files

FIGURE 6.11: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency with a dedicated-core scenario, and a tier-3 VM running an interfering workload, for 1 KiB, 100 KiB, and 1 MiB files and with varying throughput.

In Tableau, this means that for tier-2 VMs to be practical, they must have a low impact on the performance of tier-1 VMs. However, due to Tableau's novel three-level scheduler, tier-1 VMs are guaranteed service before tier-2 VMs. To illustrate this, we ran an experiment where the performance of a dedicated-core tier-1 VM was measured, while a co-located tier-2 VM ran an interfering background workload using any available idle cycles on the core. The performance of the tier-1 VM was measured using our HTTPS `nginx` workload as in previous experiments; the requests per second were varied and both latency and actual throughput were measured.

Figure 6.11 shows the results from our experiment. The figure consists of three rows and three columns. The three rows correspond to the results for different file sizes being served (1 KiB, 100 KiB, and 1 MiB, respectively), while the three columns show the throughput compared against a varying latency metric (mean latency, 99th percentile latency, and maximum-observed latency, respectively). Finally, each graph comprises four curves, one for each of four background interference workloads that the tier-2 VM was running.

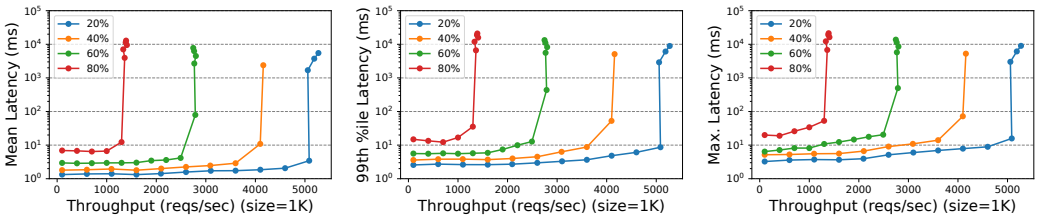


FIGURE 6.12: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency of tier-3 VM co-located with a tier-1 VM capped at varying utilizations (20%, 40%, 60%, and 80%) and running a CPU-intensive workload, for 1 KiB files and with varying throughput.

Tier-2 VMs have low impact on the performance of tier-1 VMs for the evaluated workload. As can be seen from the graphs, the variation in the performance of the tier-1 VM is negligible regardless of the characteristics of the background workload run on the tier-2 VM. In particular, given that the idle scenario (blue line in each graph) represents the baseline performance of the tier-1 VM in the absence of any interference, it can be clearly seen that its performance is comparable even in the presence of an interfering tier-2 VM on the same core. The largest reduction in performance can be seen with respect to the maximum-observed latency of 1 KiB file requests with an I/O-intensive tier-2 VM. However, note that the 99th and mean latencies are not affected, and only peak throughput is affected, that too negligibly; it is still comparable to the baseline.

We show in Chapter 7 that a highly cache-sensitive workload responds differently to cache interference, and can indeed suffer higher slowdowns as a result of tier-2 VMs. However, we note from section 6.5 that VMs under Tableau suffer the lowest slowdown when compared with other existing schedulers.

Therefore, we conclude that tier-2 VMs in Tableau can be practical, and that they do not significantly affect the performance of tier-1 VMs in dedicated-core scenarios for many practical workloads, such as the `nginx`-based web service evaluated in this chapter. For more cache-sensitive workloads, tier-2 VMs may have a higher impact (see Chapter 7), and while the impact is still low compared with other schedulers, it might make sense to avoid tier-2 VMs for such workloads (*e.g.*, via a more expensive cache- or memory-optimized offering).

It should also be noted that the performance of tier-2 VMs themselves is configurable. That is, the amount of service received by tier-2 VMs can be configured by capping the maximum utilization of tier-1 VMs, and leaving idle time for tier-2 VMs to use.

To illustrate this, we ran an experiment where the performance of a tier-2 VM was measured, while a tier-1 VM ran a CPU-bound workload, but was capped at different utilizations. The performance of the tier-2 VM was measured using our HTTPS `nginx`

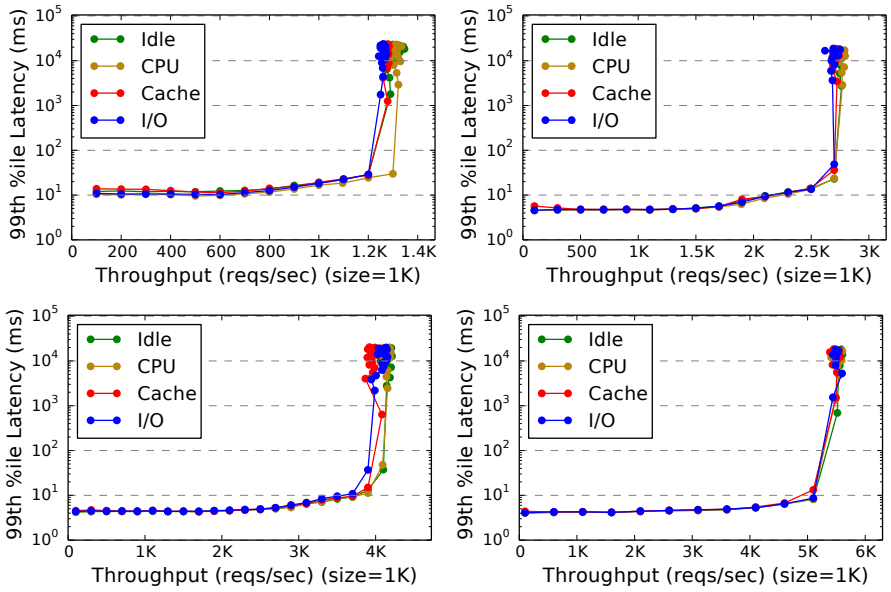


FIGURE 6.13: 99th percentile latency of a tier-1 VM capped at varying utilizations (20%, 40%, 60%, and 80%) and co-located with a tier-2 VM running various background workloads (idle, CPU-intensive, cache-intensive, and I/O-intensive), for 1 KiB files and with varying throughput.

workload while serving 1 KiB files; the requests per second were varied and both latency and actual throughput were measured.

Figure 6.12 shows the results from our experiment. The figure consists of three columns corresponding to the throughput compared against a varying latency metric (mean latency, 99th percentile latency, and maximum-observed latency, respectively). Each of the three graphs comprises four curves, one for each of four capped utilizations of the tier-1 VM (20%, 40%, 60%, and 80%).

Tier-2 VM performance can be configured by capping tier-1 VM utilization. As can be seen in the figure, the performance of the tier-2 VM depends on the amount of idle time available for it. That highest tier-2 performance can be observed when the co-located tier-1 VM is capped at 20% utilization, while the lowest tier-2 performance is observed when the tier-1 VM is capped at 80%. This, combined with the negligible impact of tier-2 VMs on tier-1 performance allow for flexible and configurable trade-offs between tier-1 and tier-2 performance.

Tier-2 VMs have low impact on the performance of tier-1 VMs. Figure 6.13 shows the performance of a tier-1 VM capped at different cappings while a co-located tier-2 background VM runs various background workloads (idle, CPU-intensive, cache-intensive, and I/O-intensive) using the `stress-ng` tool. Thus, these graphs show the performance impact on tier-1 VMs as a result of tier-2 background work. As can be seen in

Figure 6.13, the degradation in performance from tier-2 VMs on the same core is negligible. This shows that Tableau's approach to scheduling background VMs is practical and does not impact tier-1 performance significantly.

6.7 Discussion and Limitations

A rigid table-driven scheduler like Tableau is not ideal for certain scenarios. We next discuss some of the limitations of Tableau.

Lower I/O device utilization in certain capped scenarios. One of the drawbacks of a table-driven scheduler is that I/O requests are sent in periodic bursts. For example, when a VM's slot is active, it is able to enqueue packets in the network interface's ring buffer, but when the VM is preempted for a relatively long time, the network device drains its buffer and then idles. This is inefficient and results in lower throughput than a dynamic scheduler, which can ensure a more even distribution of VM execution over time, resulting in a better-utilized I/O device given the same CPU utilization.

This effect is evident in Figure 6.5(g)–(i), which shows a capped scenario with 1 MiB files where Credit achieves higher peak throughput compared to Tableau. This does not occur for smaller file sizes as they require less bandwidth (*i.e.*, utilizing the network efficiently is not so important because CPU utilization is the bottleneck when serving small files). However, for larger files, transmission time becomes significant and the VM must work harder to keep the network device sufficiently busy to ensure high throughput.

Overall, if the goal is to maximize I/O device utilization, a rigid table-driven scheduler is not ideal. However, Tableau's second-level scheduler for uncapped VMs can help with overcoming this inefficiency, as is evident in Figure 6.5(p)–(r).

Higher mean latencies in capped settings. While Tableau provides high throughput and predictable tail-latency characteristics, it can be seen in Figure 6.5(a), (d), and (g) that it performs worse compared to Credit in terms of mean latency in capped settings. This is not unexpected since capped VMs under Tableau do not have the luxury of responding to requests at any point in time; rather, they are limited to carefully controlled windows of time where they are allowed to process requests. This means that requests arriving in the blackout period between slots incur higher latencies, resulting in increased average latency. However, as can be seen in the other graphs in Figure 6.5, dynamic schedulers come with their own trade-offs, namely lower throughput in the

face of frequent scheduler invocations due to more complex, higher-overhead scheduling logic.

Semi-partitioned VM performance. In this thesis, we focus on the (common) case of fully partitioned vCPUs and do not evaluate migrating vCPUs, that is, VMs that are forced to frequently migrate across multiple cores due to semi-partitioning or localized optimal scheduling (or other table generation methods). To reiterate, we consider semi-partitioning to be rare in a controlled cloud setting as operators can pick vCPU utilizations that are easy to partition.

However, in the rare cases where semi-partitioning is unavoidable, there is undoubtedly a performance penalty to be paid by frequently migrating VMs. While Credit, Credit2, and RTDS also frequently migrate vCPUs, there is a significant difference: under these schedulers, *all* vCPUs are (non-deterministically) subject to occasional migration, so the performance penalty evens out over time. In contrast, in Tableau, migration costs are borne exclusively by vCPUs with allocations on multiple pCPUs, which is unfair.

There are several ways around this imbalance. For one, any split vCPU could be "compensated" for the increased overheads by increasing its utilization by a few percentage points and factoring this added resource usage into the cost. Alternatively, one could periodically re-generate the scheduling table to make sure that all vCPUs take a turn being split across cores. It will be interesting to explore the involved trade-offs in more detail in future work.

Other sources of unpredictability. While we focus on two sources of performance variability, CPU scheduling and cache interference, these are merely two pieces of a larger predictability puzzle. Modern server-class machines have many other sources of such performance variability, including queueing delays in shared I/O schedulers, interaction of the hypervisor and guest OS schedulers, variability arising from filesystem I/O. Case in point, in the experiments discussed in Section 6.4 we specifically removed the guest OS scheduler and disk I/O from consideration to minimize measurement noise. A complete system that comprehensively addresses all such issues is beyond the scope of this thesis, but an important challenge for future work.

Performance degradation under cache interference. As we saw in Section 6.4, there is significant performance degradation under all schedulers under interference from VMs running cache-intensive workloads. While Tableau outperforms existing schedulers, it does not attempt to mitigate the interference itself. In particular, as interference

in multi-tenant clouds is temporally localized, it may be possible to improve performance through runtime mitigation techniques. We look at this in more detail in the next chapter.

Chapter 7

Mitigating Dynamic Cache Interference

As we saw in the last chapter, cache interference from co-located VMs can result in significantly reduced throughput. However, the background workload used in our evaluation in Chapter 6 fully utilized all cores in the system entirely, and utilizations in public clouds are comparatively lower [13]. This means that, in practice, interference is lesser and non-uniform, both spatially and temporally.

Therefore, from the point of view of a VM scheduler, it is not sufficient to simply guarantee utilization and scheduling latency bounds for VMs (*e.g.*, via a scheduling table in Tableau) and expect the performance to remain consistent. Rather, a more dynamic approach must be taken to additionally mitigate changes in interference patterns at runtime.

The challenge lies in achieving this without increased runtime overheads. In this chapter, we present a novel interference mitigation mechanism that is unique to Tableau's design. Recall that under Tableau, a new scheduling table is generated every time a VM is created, torn down, or reconfigured. However, Tableau is not limited to generating tables in just these situations. Instead, in this chapter we employ an approach where tables are periodically regenerated such that they provide the same performance guarantees (*i.e.*, the utilization and scheduling latency of every VM is satisfied), but have different cache-interference characteristics.

7.1 Motivation

Cache interference is an unavoidable consequence of sharing cache memory on a given core (*e.g.*, L1 or L2 caches in Intel architectures) or socket (*e.g.*, LLC) with other tenants, and cannot be eliminated entirely. Such cache interference has a negative effect on VM performance, most notably throughput (as we saw in section 6.4).

Cache interference arises at multiple levels of the cache hierarchy: L1 and L2 caches, which are private to each core in the system, are subject to interference from VMs on the same core. Similarly, LLC interference is caused by interfering VMs running on cores that share it; in our evaluation platform, and as is common in most commercial platforms, all cores on a single socket share an LLC, and as a consequence, any VMs executing on a particular socket may cause LLC interference for co-located VMs.

Distribution of cache pressure. While cache interference, or increased *cache pressure* (or simply *interference*, as we refer to it in the rest of this chapter), is simply unavoidable when sharing caches with other VMs, it is important to note that the distribution of cache pressure may vary both temporally and spatially.

For an example of varying *temporal distribution* of interference, consider a system with 50% of VMs running a cache-intensive workload. On one end of the spectrum, 50% of all VMs may be a source of interference for 100% of the time, or at the other end of the spectrum, 100% of VMs may cause interference for 50% of the time.

Similarly, interference may be *spatially distributed* in the system, with the 50% of VMs running a cache-sensitive workload being, on one hand, uniformly distributed on all cores in the system or, in the other extreme, concentrated on a few cores.

Regardless of the precise nature of the instantaneous distribution of interference over time (*i.e.*, the temporal or spatial distribution of interference on the active cores in the system), the key point to note is that in systems that are not fully utilized, there may be room for lowering interference experienced by any particular VM at runtime by simply lowering peak cache pressure on any given core or socket. That is, we can use load balancing to redistribute cache pressure evenly across the system so that no specific VM is unduly penalized.

To demonstrate this, we show PARSEC benchmark [15] results under two different cache interference scenarios.

The first (CI-1) runs a cache-intensive workload on a specified number of randomly-chosen VMs in the system, and the random subset of VMs is changed every ten seconds. Since the VMs are chosen randomly, interference is uniformly distributed over

all active cores in the system in the long run. Therefore, CI-1 represents the best-case performance degradation in the system for a given number of interfering VMs.

The second interference workload (CI-2) runs a cache-intensive workload on a specified number of VMs, but specifically targets cores on the same socket as the VM being evaluated. The workload periodically (every second) determines the placement of the evaluation target VM using the current scheduling table, and repartitions the interfering VMs successively on to the same core as the target VM, the same socket as the target VM, and finally other sockets in the system. This serves as the other extreme of performance degradation, where a specific VM is targeted explicitly, and possibly maliciously, with the intent of degrading its performance.

Experimental setup. We ran the `canneal` benchmark, which is a part of the PARSEC benchmark suite [15]. It was chosen as it is a cache-aware benchmark whose performance is well-known to be sensitive to cache interference. The `canneal` benchmark was run on our evaluation target VM, and the total time to completion measured. A subset of the remaining VMs, of varying size, ran a cache-intensive stress workload using the `stress-ng` tool. A hundred samples were collected for each experiment using the `simsmall` dataset provided by PARSEC. We calculated 95% confidence intervals for each experiment using bootstrapped confidence intervals [40] and plot them for each bar presented.

Figure 7.1 shows the results of the `canneal` benchmark under varying number of VMs running both types of interference workloads (CI-1 and CI-2), which are discussed below.

Figure 7.1 shows a bar chart where the X-axis shows the number of VMs running the interfering workload and the Y-axis shows the slowdown in performance compared to a system where no VMs are running an interference workload. The blue and red sets of bars show the results for CI-1 and CI-2, respectively.

As can be seen from Figure 7.1, as the number of VMs running the cache-interfering workload is increased, the performance degrades. In particular, it can be clearly seen that the performance degradation is more severe under the targeted interference (*i.e.*, in scenario CI-2). This is not surprising as the targeted CI-2 interference is designed to maximally interfere with the VM being evaluated.

Note that the performance under randomly distributed interference (CI-1) represents the best-case scenario. In contrast, the performance under CI-2 represents the worst-case scenario.

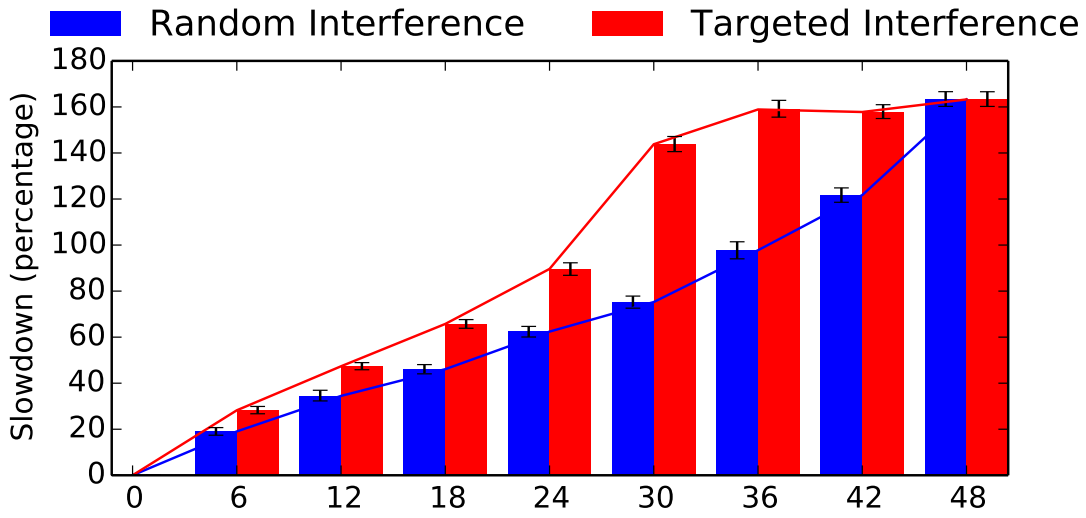


FIGURE 7.1: A comparison of `canneal` performance running in a VM in a high-density scenario under varying number of interfering VMs running two types of interference workloads.

When interference is uniformly distributed across all cores in the system, it minimizes the peak pressure. That is, no single core experiences undue, targeted cache pressure. Cache pressure *with respect to a specific core in the system*, may be lowered below this best-case level (*e.g.*, if all cache pressure is targeted on a different socket than the one being measured). However, this comes at the cost of fairness, as some other VM will receive higher peak pressure. Alternatively, another approach that lowers cache pressure would involve isolating all VMs running cache-intensive workloads onto a separate socket, but again this sacrifices fairness as each of them experience undue cache pressure compared to those VMs not running cache-sensitive workloads. Therefore, scenario CI-1 represents the best-case cache-interference achievable on a multi-tenant machine without sacrificing fairness and penalizing specific VMs.

On the other hand, a specific VM in the system experienced maximal cache pressure when all VMs running cache-intensive workloads are co-located on, successively, the same core as the VM, the same socket, and finally, on other sockets in the system. This is the behavior of the CI-2 scenario, which for a given number of interfering VMs, maximizes the interference experienced by the evaluation target VM.

Therefore, the gap between the blue and red *lines* in Figure 7.1, both of which represent the same data as the bars except as a line plot, represent the potential for improvement via load balancing. Ideally, we want to be able to achieve slowdowns in VM performance approaching that under CI-1 (*i.e.*, the blue line), while being subject to CI-2 interference (*i.e.*, the red line).

We now present the design, implementation, and evaluation of two mitigation strategies that we implemented in Tableau. The first approach randomizes schedules continuously so as to prevent interference hotspots from forming at runtime. The second strategy attempts to detect specific VMs causing cache interference using low-level, per-VM performance counter data, and then redistribute interfering VMs by regenerating new, optimized scheduling tables.

7.2 Mitigation Strategy 1: Random Repartitioning

As we saw in Figure 7.1, the `canneal` benchmark performs better when the interference is spread out across all cores in the system as opposed to being concentrated on a few targeted cores.

The intuition behind the first mitigation strategy is that, while we cannot convert CI-2 interference into CI-1 interference by modifying the interference characteristics of VMs themselves, we can achieve the same effect by randomizing the placement of VMs in the system.

By randomizing VM placement at runtime, this strategy prevents any interference hotspots from building up on specific cores, and dissipates the interference across all cores in the system when averaged over longer intervals of time.

To test this, we implemented a userspace *randomizer* daemon that takes the current scheduling table and swaps the slots of two randomly-chosen VMs. This process was repeated one hundred times to ensure sufficient randomization. The new table was then pushed to the hypervisor. Finally, this entire process was performed continuously in a loop, with no delay between iterations, while the `canneal` workload was evaluated.

This highlights one limitation of the randomized approach, which requires many interchangeable, similarly-configured VMs that can be repartitioned at runtime. However, we note that this is not a major issue for two reasons. First, our approach of swapping VM allocations is only one way to repartition VMs. An alternative approach, for example, would involve modifying the partitioning heuristic used in the planner to be less deterministic. This would allow for regenerating randomized tables directly with the planner. Second, from the perspective of a cloud provider, co-locating similarly-configured VMs on the same machine is not problematic. Therefore, we consider our randomized approach to be practical.

Figure 7.2 shows the results of an experiment where thirty VMs were generating CI-2

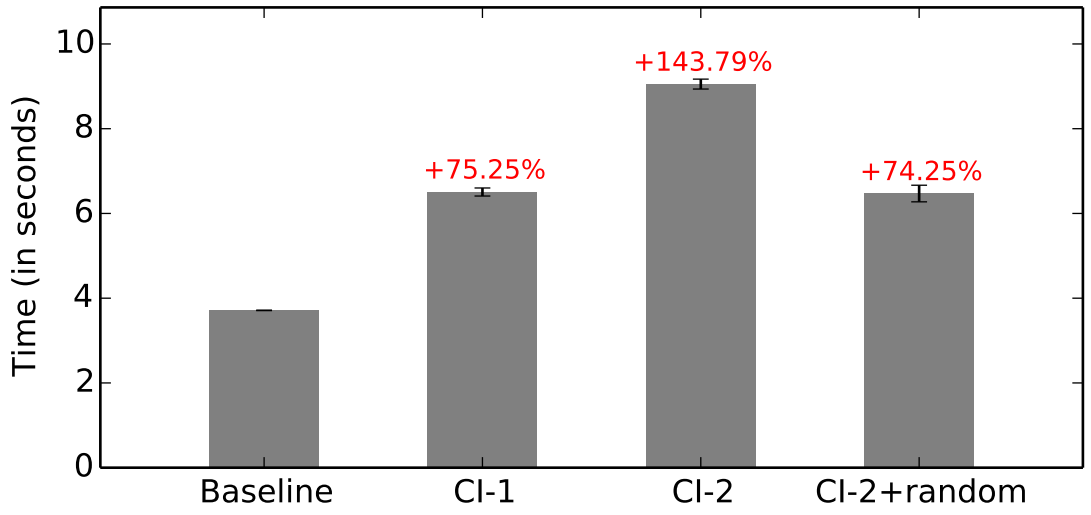


FIGURE 7.2: A comparison of `canneal` performance running in a VM under a no-interference scenario (Baseline), with thirty CI-1 VMs, thirty CI-2 VMs, and thirty CI-2 VMs with randomized mitigation

interference and our randomizer is running. The number of interfering VMs was chosen to be thirty as Figure 7.1 shows the largest gap in performance between CI-1 (random) and CI-2 (targeted) interference for thirty VMs. We compare against a baseline of performance when all VMs are running CPU-intensive work without a significant cache footprint.

Figure 7.2 shows the results in the form of a bar graph where the Y-axis shows the mean execution time of the `canneal` benchmark, and the X-axis shows four bars: a baseline where all VMs run a CPU-bound loop, thirty VMs running a CI-1 workload, thirty VMs running a CI-2 workload, and finally, thirty VMs running a CI-2 workload while the randomizer daemon is active. The values above the bars indicate the percentage slowdown compared to the baseline.

First, it is evident from Figure 7.2 that CI-2 interference results in significantly longer completion times compared to CI-1 interference (an additional 143% under CI-2 compared to the baseline, as opposed to a 75% increase under CI-1).

Next, we observe that `canneal` performance under a targeted CI-2 interference with our randomizer active achieves comparable performance as when under random interference (CI-1). This is because, by randomizing tables and consequently VM placement, we achieve the uniform interference distribution of a CI-1 workload even though the VM is under CI-2 interference.

In fact, we found that `canneal` performance with the randomizer enabled, and targeted CI-2 interference was comparable or better than the performance under random CI-1 interference.

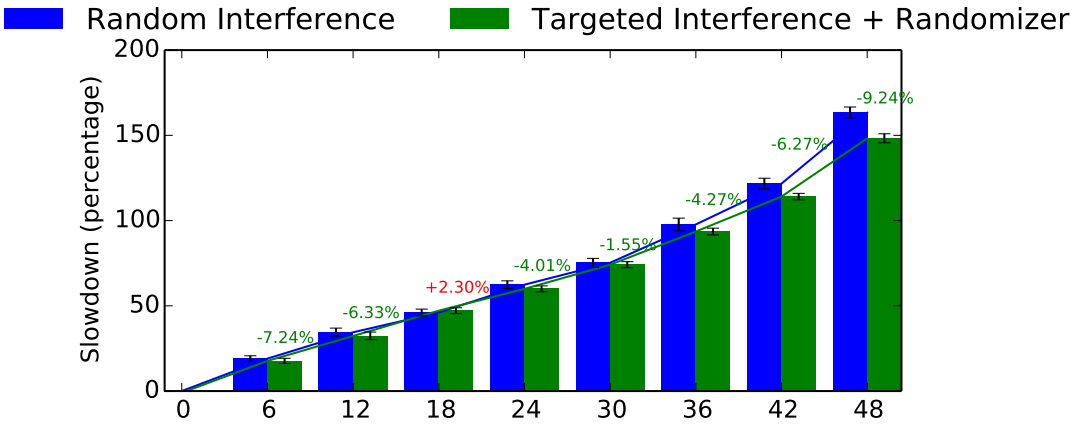


FIGURE 7.3: A comparison of `canneal` performance running in a VM in a high-density scenario under varying number of interfering VMs in two scenarios: one with random CI-1 interference, and one with CI-2 interference but our randomizer active.

Figure 7.3 shows a bar chart similar to Figure 7.1, where the X-axis shows the number of VMs running the interfering workload and the Y-axis shows the slowdown in performance compared to a system where no VMs are running an interference workload. The blue and green sets of bars show the results for CI-1 and CI-2 with our randomizer active, respectively.

As can be seen in Figure 7.3, `canneal` performance under CI-2 interference with our randomizer active is either comparable or better than its performance under CI-1 interference. The reason for this has to do with an asymmetry in our setup. While our evaluation machine has two sockets with eight cores each, four of the cores on the first socket are dedicated to Xen's dom0. As a result, the peak number of interfering VMs that can be present on the first socket is limited to half that on the second one. The results in Figure 7.1 and Figure 7.2 were generated with the evaluated VM being partitioned onto a core on the second socket (*i.e.*, a socket whose LLC is susceptible to interference from all eight cores on the socket in the worst case). In contrast, the randomizer migrates VMs across all cores, including those of the first socket, resulting in lower interference on average. This highlights an advantage of the randomized mitigation strategy: by migrating VMs across all cores in the system, it is able to take advantage of imbalances in the levels of LLC interference on different sockets in the system, resulting in improved average performance.

However, we note that for a more balanced setup **(i)** where all cores on each socket are avaible for guest VMs, or **(ii)** where dom0 cores are equally distributed among all sockets in the system, the randomized approach can be expected to achieve performance only comparable to that under CI-1 interference.

Based on our observations, we conclude that continuous random repartitioning of VMs at runtime is an effective strategy for lowering peak cache pressure and distributing it uniformly across the entire system.

7.3 Mitigation Strategy 2: Load Balancing Based on Performance Counters

An alternative mitigation strategy that we evaluate involves monitoring the performance of VMs at runtime, detecting when they perform cache-intensive work, and using this information to balance the pressure across sockets and cores in the system so as to avoid hotspots of cache pressure.

Conceptually, the advantage of this strategy over the randomized approach is that fewer VMs need to be moved around at runtime, requiring fewer resources for the table regeneration, as well as reducing the number of VMs that are migrated to a different core than the one on which they are currently running.

One way to achieve this would be via runtime monitoring and load balancing within the hypervisor scheduler itself. However, we do not implement this approach as it would result in higher runtime overheads of the scheduler, which as we have shown in Sections 6.2 and 6.4, lowers throughput.

Rather, we take an alternative approach: we only gather lightweight performance monitoring metrics at runtime within the hypervisor. These are then extracted by an asynchronous userspace daemon, which uses them to detect cache-intensive VMs. Finally, the userspace dameon rebalances cache-intensive VMs evenly across all cores in the system in order to lower hotspots of cache pressure. This rebalancing is performed by pushing a new, optimized table to the hypervisor.

We first describe how we gather metrics in a lightweight manner, without incurring significant runtime overhead.

Monitoring per-VM performance counters. As discussed in section 2.6, most modern processors used in datacenters come with a performance monitoring unit (PMU) that can be used to count low-level architectural events.

We extended Tableau to simultaneously monitor multiple performance counters for each vCPU in the system. Our evaluation platform (described in section 6.4), supported four programmable counters that could be counted simultaneously at any given time.

The implementation was straightforward: each vCPU-specific structure was extended with accumulators for each counter being measured. When a particular vCPU is dispatched, the dispatcher activates the performance monitoring unit to begin counting a set of (configurable) events, and when the vCPU is de-scheduled, it stops monitoring them. The observed count of each event are added to an event-specific accumulator within the per-vCPU structure.

Apart from the mechanism to collect performance-monitoring data for each vCPU, a custom hypercall command was implemented to allow a userspace daemon to read out the accumulator values for each vCPU in the system.

Programming the PMU. The added overhead in the system is not evaluated but is low as programming the PMU involves only two lightweight `wrmsr` instructions for each counter being monitored, and reading out the current value requires just a single `rdmsr` instruction for each counter being monitored.

Intel x86 architectures provide special *model-specific registers* (MSRs) which can be used to control various auxiliary functionalities such as performance monitoring. Reading and writing MSRs to and from a specified address, respectively, is done using the special `rdmsr` and `wrmsr` instructions.

By default, Tableau counts four events: the number of cycles a VM executes for, the number of instructions retired, the number of last-level cache misses, as well as the number of last-level cache references.

With these counters available for each VM in the system, we attempt to detect the subset of VMs currently performing cache-intensive work.

Performance monitoring daemon. We implemented a userspace daemon (similar to the tier-2 load balancing daemon described in section 4.3) that extracts performance counter data for each VM in the system and streams it to a detection and mitigation framework.

The detection part of the framework is responsible for determining when VMs start or stop performing cache-intensive work. The mitigation part is responsible for generating new, optimized tables taking into account the current state of the system.

The detection framework has been designed to keep a record of timeseries performance counter data for each VM, core, and socket in the system. Using this timeseries data, it efficiently tracks trend lines using trailing simple moving averages over configurable time intervals. The general mechanism for detecting cache-intensive work uses a pair of trailing moving averages, one over a short time interval and another over a longer

time interval, to determine the current phase of the VM. The trend line for the shorter time interval, owing to the shorter time window used for averaging, is more responsive to short-term changes in VM behavior compared with the trend line for the longer time interval, which is less susceptible to variations in short-term, volatile changes in VM behavior. As a result, we can detect a phase change (*e.g.*, a VM starts doing cache-intensive work) when the trend line for the shorter interval exceeds the trend line for the longer interval by a certain (configurable) threshold. Similarly, we can also detect the reverse phase change (*e.g.*, a VM stops running cache-intensive work) when the trend line for the shorter time interval falls below that for the longer interval by another (configurable) threshold.

We discuss two metrics for detecting such phase changes below: one using LLC misses, and one using cycles per instruction (CPI).

Detecting cache interference using LLC misses. The simplest approach to detecting cache-intensive work is to track the LLC misses of the VM. This value has been observed to increase sharply when a VM begins to perform cache-intensive work.

To demonstrate this, we performed an experiment where a single VM was monitored while it went through three phases: it was idle for the first sixty seconds, then ran a cache-intensive stress benchmark, using the `stress-ng` tool, for the next sixty seconds, and finally ended with another idle phase sixty seconds long. Figure 7.4 shows various trend lines tracking LLC misses over different window sizes for the evaluated VM, and shows how they respond to VM workload changes.

As can be seen from Figure 7.4, there is a visible rise in the trend line at sixty seconds when the cache-intensive benchmark is started, as well as a drop at 120 seconds. It can also be seen how the choice of time window affects the volatility and responsiveness of the trend. In particular, a shorter time window results in more volatile but more responsive trends, while a longer one results in less responsive, but more stable trends.

We also ran the same experiment, but instead of starting a cache-intensive stress benchmark, we ran a CPU-intensive one instead. As this benchmark is a simple loop and does not have a significant cache footprint, it serves as a test of how useful LLC misses are as a metric for detecting cache-intensive work.

As can be seen from Figure 7.5 there is a slight increase in the LLC misses at 60 and 120 seconds. This is because although the workload used is CPU-intensive, it still has some cache accesses, albeit significantly lower than for Figure 7.4, which used a cache-intensive workload. Therefore, in order to robustly classify VMs running cache-intensive workloads, but not CPU-intensive ones, a higher threshold can be configured in the detection framework.

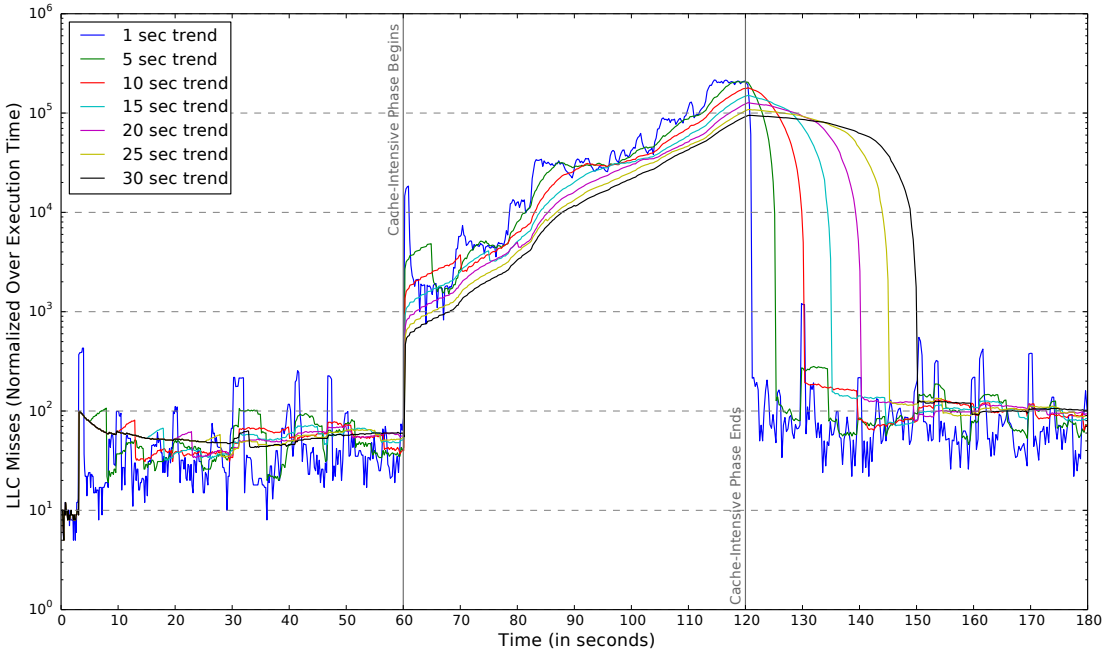


FIGURE 7.4: Trend lines tracking LLC misses for a single VM and varying time windows. The VM goes through three phases of sixty seconds each: idleness, CPU-intensive work, and again idleness.

Detecting interference using cycles per instruction. A second metric we explored for determining when interference occurs was the well-studied *cycles-per-instruction* (CPI) metric [130].

CPI is a measure of VM performance that looks at the average number of cycles it takes to retire an instruction. That is, the number of cycles a VM takes until an instruction is completed and its results are committed in the architectural state of the system. In general, CPI increases as a result of increased cache pressure, as instructions take longer to retire due to longer memory fetches holding up their completion.

CPI is measured by calculating the ratio of the number of unhalted CPU cycles (*i.e.*, cycles where the VM was executing on the core) to the number of instructions retired, both of which can be monitored accurately in the Intel architecture using low-level performance counters.

The problem with cycles per instruction. Unfortunately, using CPI to detect cache-interference in a VM environment has some downsides. Recall that CPI is a ratio between cycles executed and instructions retired. While it is meaningful to compare CPI across VMs if one of either the numerator or denominator are changing, this becomes an issue if both are changing at runtime.

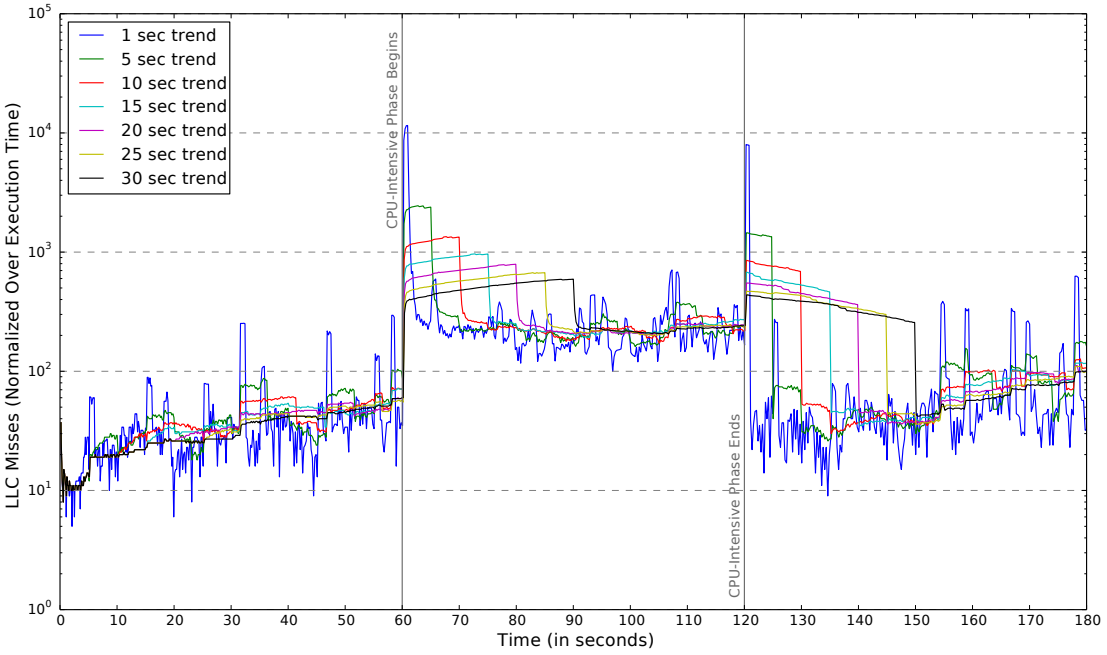


FIGURE 7.5: Trend lines tracking LLC misses for a single VM and varying time windows. The VM goes through three phases of sixty seconds each: idleness, CPU-intensive work, and again idleness.

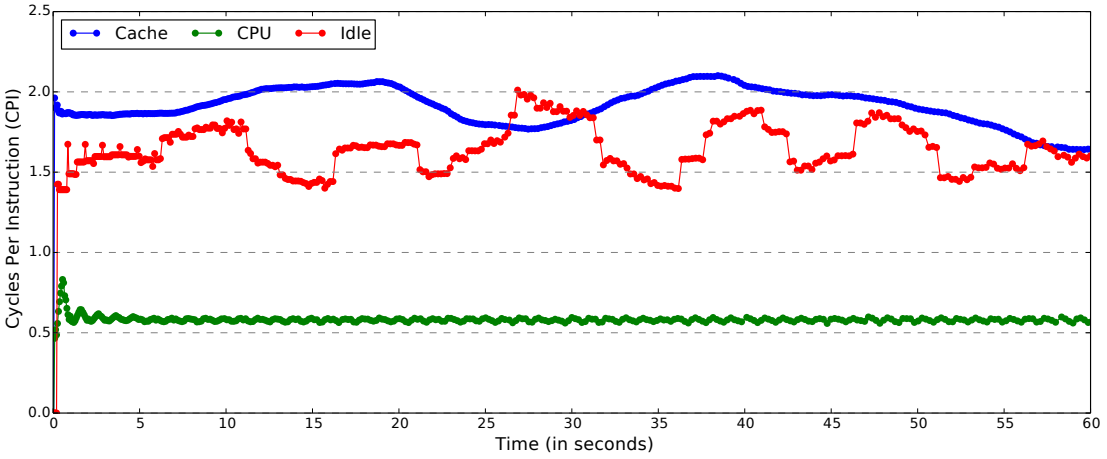


FIGURE 7.6: A comparison of CPI over a five second interval, measured for a VM when it is idle, when running a CPU-intensive loop, and when running a cache-intensive workload

In particular, the numerator, the number of cycles executed, depends on the idleness of a VM at any given time. On the other hand, the denominator changes based on both the idleness of the VM as well as how much interference the VM is experiencing at any given time. This makes it difficult to compare across VMs that have different idle time characteristics in order to compare the interference experienced by each.

Figure 7.6 shows the CPI over a five second interval for a VM when it is idle, when running a CPU-intensive loop, and when running a cache-intensive workload. As can

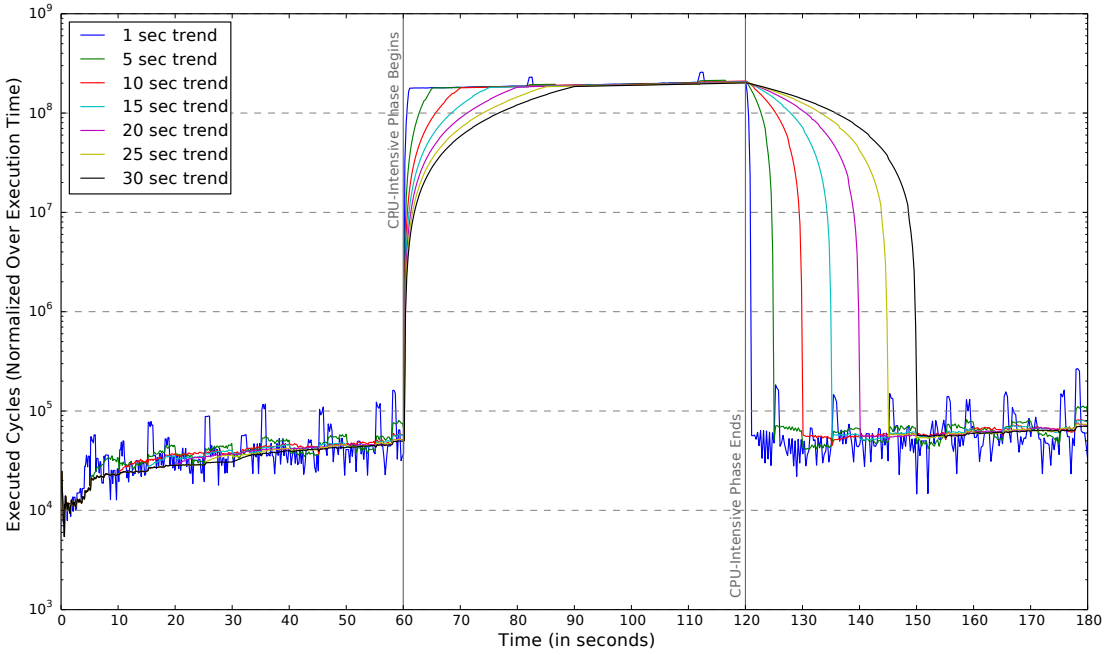


FIGURE 7.7: Trend lines tracking number of cycles executed by a single VM and varying time windows. The VM goes through three phases of sixty seconds each: idleness, CPU-intensive work, and again idleness.

be seen in the figure, the curves when running a CPU-intensive and cache-intensive workload show the correct trends. For CPU-intensive work, as there are fewer memory fetches, there is high instruction-level parallelism resulting in less than one cycle per instruction retired. On the other hand, for the cache-intensive workload, we see it taking greater than one cycle to retire a single instruction. While these two are comparable, the problematic case is the curve where the VM is idle. In this case, both the denominator and numerator change compared with the CPU-intensive and cache-intensive cases, making it incomparable.

The second issue with CPI is that it does not pinpoint the source of interference accurately. A rise in CPI for a single interfering VM is accompanied with a rise in CPI for other VMs sharing resources with it (*e.g.*, other VMs on the same socket sharing the LLC with the interfering VM). As a result, while CPI can be used to establish *that* VMs are experiencing interference, we cannot use it to robustly pinpoint the specific VM that is the source of the interference.

Since **(i)** acting on the CPI of near-idle VMs does not yield meaningful results, and **(ii)** they are not a source of interference anyway, one alternative is to additionally detect when VMs become idle or busy. We can then filter idle VMs when determining the set of interfering VMs. Figure 7.7 presents a graph showing how trend lines tracking the number of cycles executed for in a given time interval can be used to determine when a VM becomes busy or idles.

Figure 7.7 shows trend lines tracking the number of cycles a VM executed for while it underwent three phases of execution of sixty seconds each: idleness, CPU-intensive work, and again idleness. As can be seen in Figure 7.7, the count of executed cycles can be used to accurately detect the idle and busy phases of a particular VM.

However, due to **(i)** the issue of correlated CPI rise, and **(ii)** the fact that it is difficult to track accurate timeseries data for VMs (excluding idle intervals) when they are constantly idling, we did not explore the use of CPI further. Rather, we chose to use the raw number of LLC misses to detect interfering VMs instead.

To reiterate, our general approach to detecting interference involves configuring two trend lines, each with a window size and two thresholds, and detecting when the trend for the shorter time interval exceeds or falls below the configured threshold, compared to the trend line for the longer time interval. We discuss our selection of the parameters for configuring the pair of trend lines next.

Tuning trend line parameters. There is significant opportunity for finding ways to choose both the window size as well as the two threshold values, but this is beyond the scope of this thesis. Our goal is to design mechanisms that allow for implementing arbitrary policies, and presenting proof-of-concept policies that shows the benefit of the approach. In our detection, we tested a range of parameters and settled for a window length of one second for the short trend, a window length of five seconds for the long trend, and a threshold of 1,000 LLC misses.

Mitigation strategy. Following the detection of VMs performing cache-intensive work, the goal of the mitigation algorithm is to distribute the interference evenly across all cores in the system. In particular, it is not our goal to sacrifice fairness in order to improve performance (*e.g.*, by co-locating all cache-intensive VMs together on a separate socket). While this would improve performance for other VMs compared to a uniformly-distributed interference, it would also unfairly penalize a subset of tenants. Rather, our goal is to *minimize peak interference*.

Figure 7.8 shows the performance of PARSEC's `canneal` workload for two scenarios: when all VMs on a particular socket generate cache interference (using the `stress-ng` benchmark tool), and when the same number of VMs are spread across two sockets equally. Both are compared with a baseline performance where all VMs are running a CPU-intensive workload without significant cache footprint.

As can be seen from Figure 7.8, splitting VMs evenly across two sockets unsurprisingly lowers cache contention and improves performance (*i.e.*, from 9.5 seconds on average to

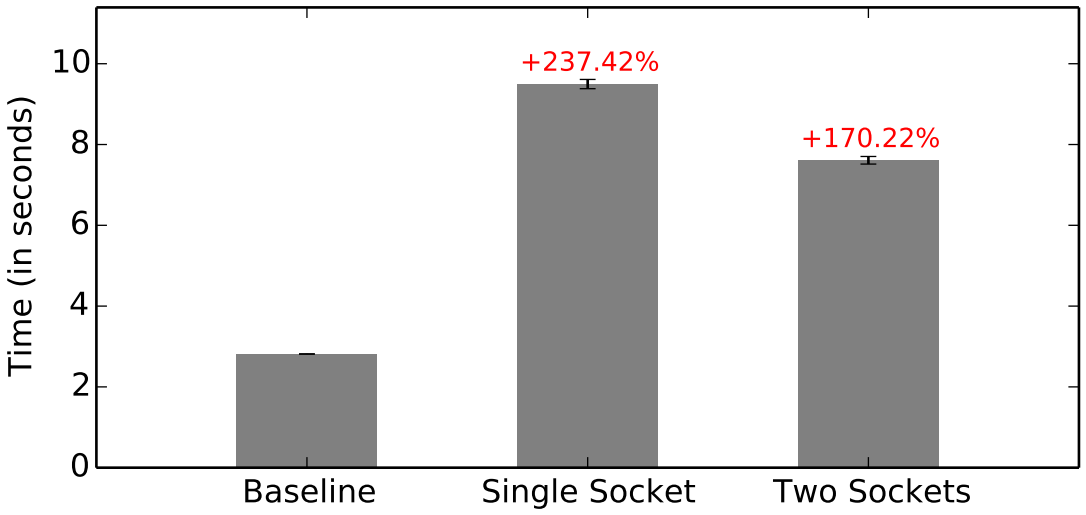


FIGURE 7.8: A comparison of `canneal` performance running in a VM under **(i)** a no-interference scenario (Baseline), **(ii)** with all VMs on a socket causing cache interference, and **(iii)** with the same VMs spread across both sockets in the system.

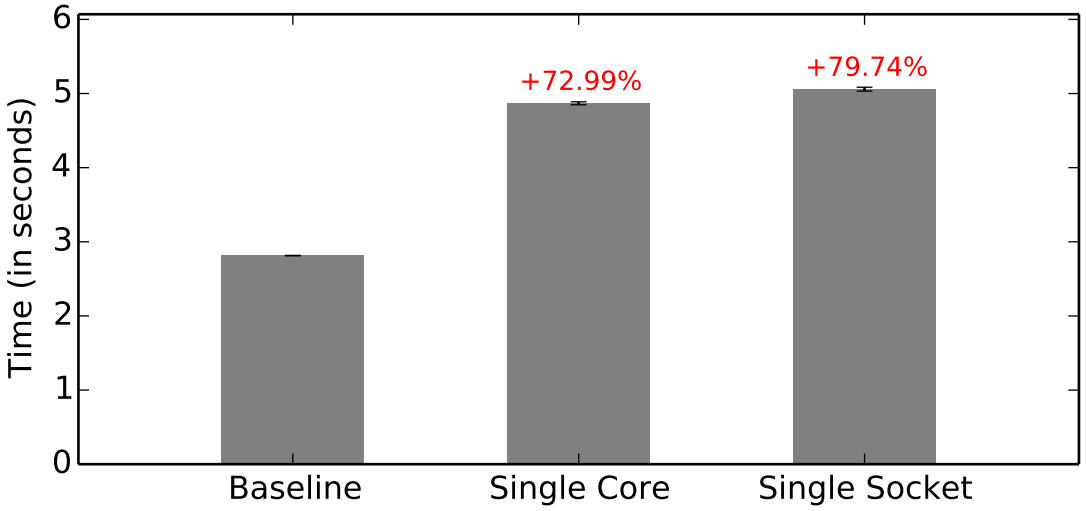


FIGURE 7.9: A comparison of VM performance under a no-interference scenario (Baseline), with three interfering VMs are co-located on the same core as the evaluated VM, and when the three VMs are spread across cores of the same socket.

7.6 seconds). This is because each socket has its own LLC, and spreading the interfering VMs across two sockets instead of one results in lower peak interference per socket.

Similarly, Figure 7.9 shows the performance of `canneal` running on one of four VMs placed on a single core versus when the four VMs are spread across the cores of a single socket.

Interestingly, it can be seen in Figure 7.9 that there is a slight performance improvement when co-locating interfering VMs onto a single core, compared to when the four VMs

are spread out across the socket socket (only a 73% degradation in the former compared to the baseline, versus an 80% degradation in the latter case). This is likely due to the specific scheduling table created where, when all VMs are on a single core, no parallelism is possible, while when spread across a socket, VMs can interfere simultaneously with each other, resulting in increased cache pressure. However, we note that this does not generalize to a principle that says co-locating cache-intensive VMs on few cores is preferable. Rather, it depends on the specific schedule, and it can be beneficial to spread the interference across an entire socket.

Figure 7.9 shows that interference concentrated on a single socket is less preferable than when concentrated on multiple sockets. We also conclude that, independent of specific scheduling tables, concentrating interference on a particular core is less preferable to spreading it out over all the cores in the same socket. That is, as a general principle, reducing peak interference on **(i)** sockets, and **(ii)** cores within each socket is a good way to reduce cache interference experienced by VMs without causing undue performance degradation on specific VMs (*i.e.*, by sacrificing fairness).

Mitigation algorithm. Our mitigation algorithm proceeds in two phases. In the first phase, the number of cache-intensive VMs are split evenly across all sockets in the system, owing to the fact that each socket has its own LLC. This is done by each socket in the system first donating excess cache-intensive VMs, followed by all sockets with fewer cache-intensive VMs picking them up based on their deficit.

Once this is complete, the number of cache-intensive VMs are equally split across all the sockets in the system. The second phase then uses a similar technique to evenly distribute cache-intensive VMs across cores within the socket. That is, VMs from cores with an excess number of cache-intensive VMs get reassigned to cores with a deficit in cache-intensive VMs.

At the end of the two phases, the cache-intensive VMs in the system are spread out evenly across all sockets in the system

Therefore, the cache pressure in any given socket is comparable to that of any other socket in the system, with no single socket receiving undue cache pressure. Similarly, the interference per core within each socket is comparable to that of other cores in the socket. This configuration is written as a configuration file, which is used by the userspace daemon to generate and push a new scheduling table to the hypervisor periodically (every second in our experiments).

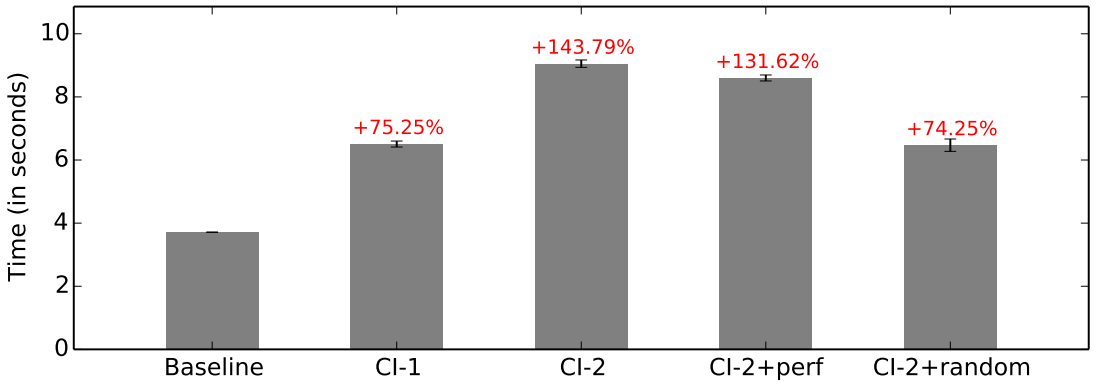


FIGURE 7.10: A comparison of VM performance under a no-interference scenario (Baseline), with thirty CI-1 VMs, thirty CI-2 VMs, thirty CI-2 VMs with performance-counter-based mitigation, and thirty CI-2 VMs with randomized mitigation

Experimental results. We measured the performance of PARSEC's `canneal` benchmark under CI-2 interference with thirty interfering VMs, and with our performance-counter-based mitigation strategy enabled.

Figure 7.10 shows the results in the form of a bar graph where the Y-axis shows the mean execution time of the `canneal` benchmark, and the X-axis shows five bars: a baseline where all VMs run a CPU-bound loop, thirty VMs running a CI-1 workload, thirty VMs running a CI-2 workload, thirty VMs running a CI-2 workload while the performance-counter-based mitigation is active, and finally, thirty VMs running a CI-2 workload while the randomized mitigation is active. The values above the bars indicate the percentage slowdown compared to the baseline.

As can be seen in the figure, with 30 VMs generating CI-1 interference, we see an increase of approximately 75% in completion time for the `canneal` benchmark. Similarly, with 30 VMs generating CI-2 interference, we see a 143% increase.

With our performance-counter-based mitigation strategy enabled, we see improved performance compared to no mitigation strategy (only a 131% increase compared with 143%). However, it does not perform as well as the randomized strategy discussed in section 7.2, which results in performance comparable to that under CI-1 interference, while experiencing targeted CI-2 interference.

This is primarily due to a lack of robustness in our detection mechanism for pinpointing VMs performing cache-intensive work. During the course of running experiments, we observed significant noise in the detection mechanism with lots of false positives, indicating that our performance-counter based detection technique is simply not robust enough for the purpose of detecting interfering VMs. However, the false-positive rate

can be lowered by employing more sophisticated phase detection techniques for time-series data, which is beyond the scope of this thesis. However, we note that detection techniques arising from future work can be easily integrated into the current detection framework presented in section 7.3.

Discussion. There are two major areas for improvement for the performance-counter-based mitigation technique. The first involves exploring more robust mechanisms for detecting VMs running cache-intensive workloads. The detection mechanism presented in section 7.3 results in high false positive rates resulting in unnecessary table changes. Second, the tuning parameters for the trend lines can have a significant impact on performance, but are currently chosen based on limited experimental observations. Better approaches to determining the right tuning parameters can result in improved mitigation.

In this chapter, we showed how dynamic changes in the interference characteristics of VMs can result in large variability in performance. We also showed how such variability can be reduced in a semi-offline fashion in Tableau using two strategies: one where VMs are randomly load balanced so as to prevent cache pressure from being concentrated in certain parts of the system, and one using performance monitoring data to detect cache-intensive VMs and distribute them uniformly across sockets and cores in the system. While the performance monitoring based approach does not provide significant benefit due to the lack of a robust detection mechanism, the randomized approach maintains performance comparable to a randomly-generated interference (CI-1), even when under targeted interference aimed at maximizing cache pressure (CI-2).

Chapter 8

Conclusion

In this thesis, we presented the design and implementation of Tableau, a novel VM scheduler for public clouds. Tableau comprises a three-level scheduler, where the first level, which uses a table-driven scheduler coupled with an asynchronous userspace planner, enables capped VMs with strong guarantees on utilization and scheduling delay. The second level further enables uncapped VMs, which execute beyond their resource limits specified in the table to use up any additional idle time on the core. Finally, the third level enables a lower-priority tier of VMs to execute using idle cycles in the system with a low impact on the performance of table-driven VMs.

We presented an extension of Tableau to deal with changing cache interference at runtime. Our basic approach involved regenerating optimized tables asynchronously based on two mitigation techniques. The first technique was a randomized approach, where VM placement was periodically changed at runtime by repartitioning all VMs in the system. The second technique was a performance-counter-based approach that detected interfering VMs via low-level performance counter data, and reduced cache pressure by distributing interfering VMs uniformly across the entire system.

Below, we briefly summarize key results pertaining to Tableau and present open questions to be explored in future work.

8.1 Summary of Results

Our evaluation of Tableau showed the following:

1. The time and space overheads of Tableau's planning step are acceptable relative to typical VM commissioning and decommissioning times.
2. Tableau incurs low scheduling overheads compared to other Xen schedulers.

3. Tableau offers both predictability (*i.e.*, consistent, low latencies) and high throughput in a high-density scenario compared to other Xen schedulers.
4. Tableau provides comparable or higher throughput for VMs compared to existing schedulers when configured with dedicated cores for each VM.
5. Tableau can be configured to provide performance guarantees for tier-2 background VMs.
6. Tier-2 VMs have a low impact on the performance of tier-1 VMs for the evaluated workload.

Our evaluation of the cache-mitigation extension to Tableau showed that:

1. There is a significant difference between the performance of VMs when experiencing random interference versus targeted interference (with the latter aimed at maximizing interference). We conclude that there is scope for a dynamic load balancing approach to mitigate peak cache interference.
2. A randomized mitigation strategy results in performance comparable to random cache interference even when experiencing worst-case targeted interference.
3. The performance-counter based mitigation strategy results in improved performance compared to targeted interference, however performs poorly compared with the randomized mitigation strategy. We attribute this primarily to lack of a robust technique for detecting interfering VMs from low-level performance counter data.

In conclusion, we believe that Tableau provides a high-performance, and predictable alternative to existing VM scheduler designs used in public clouds. In addition, the better isolation between SLA-backed VMs and background VM under Tableau opens up potential for improved server utilization in cloud datacenters. Finally, the flexible semi-offline approach allows for rapidly extending the scheduler using high-level languages, tools, and libraries, while maintaining low runtime overheads.

8.2 Open Questions and Future Work

In this section, we present some open questions pertaining to Tableau to be explored in future work.

Planner improvements. While the planner implementation presented in this thesis performs acceptably, performance of table generation can be improved significantly. First, the Python-based implementation can be replaced with an implementation in a faster language such as C or C++. Second, the writing of tables to disk as an intermediate step can be eliminated entirely. Finally, tables can be generated incrementally by comparing against the current table and rebuilding only parts that need to be changed. An alternative, when VM sizes are known upfront, would be to allow for pre-generating slots in the system to allow for fast VM creation and teardown without any planning delay. This would potentially enable on-demand, container-style VM lifetimes with stronger isolation than containers alone can provide.

Robust detection of cache-intensive VMs. As seen in Chapter 7, the design of a robust technique for detecting VMs performing cache-intensive work using low-level performance counter data is still an open area for further study.

Optimizing Tableau tables for secondary performance characteristics. Our work on mitigating cache interference is a specific instance of a general class of solutions where tables are generated that have the same utilization and scheduling latency guarantees, but differ in some secondary characteristic (*e.g.*, distribution of LLC pressure). On area of future research involves exploring other secondary characteristics that tables can be optimized for. As an example, I/O performance can vary across different cores in the system due to differences in proximity to the I/O device, and regenerating tables at runtime that place I/O-bound VMs on cores closer to the devices would improve I/O performance.

End-to-end guarantees for distributed applications. While Tableau aims to provide guarantees for a VMs on a single machine, modern applications may have more complex architectures distributed across multiple machines. For example, multi-tier, microservice-based web applications may consist of multiple services distributed across different machines working together to produce a response. Therefore, one potential direction for future research would involve extending Tableau to provide stronger guarantees for distributed application architectures running on VMs on different machines. A further extension of this direction would be to design a complete system that combines Tableau with other orthogonal isolation techniques (*e.g.*, the cluster scheduler, the datacenter network) to provide stronger end-to-end guarantees.

Appendix A

Extended Evaluation: Effect of Varying Scheduling Latency

This section shows how the performance of each of the four evaluated Xen schedulers (Credit, Credit2, RTDS, and Tableau) varies with different settings of scheduling latency. The details of the experimental setup are described in detail in section 6.4.

The results presented validate the choice of parameters for Credit in our evaluation presented in Chapter 6. In particular, we find that with an idle background, Credit performs best with a 1ms global timeslice. However, in the presence of a background workload (*e.g.*, CPU intensive, cache intensive, or I/O intensive), it performs the worst of all evaluated values. Either a 5ms or 10ms timeslice tends to perform best in the majority of evaluated scenarios, and our experiments presented in Chapter 6 use a 5ms timeslice.

Since there is no way to configure the scheduling latency under the Credit2 scheduler, the graphs for it are omitted.

A.1 Credit Scheduler

A.1.1 Idle Background Workload

A.1.1.1 Capped Scenario

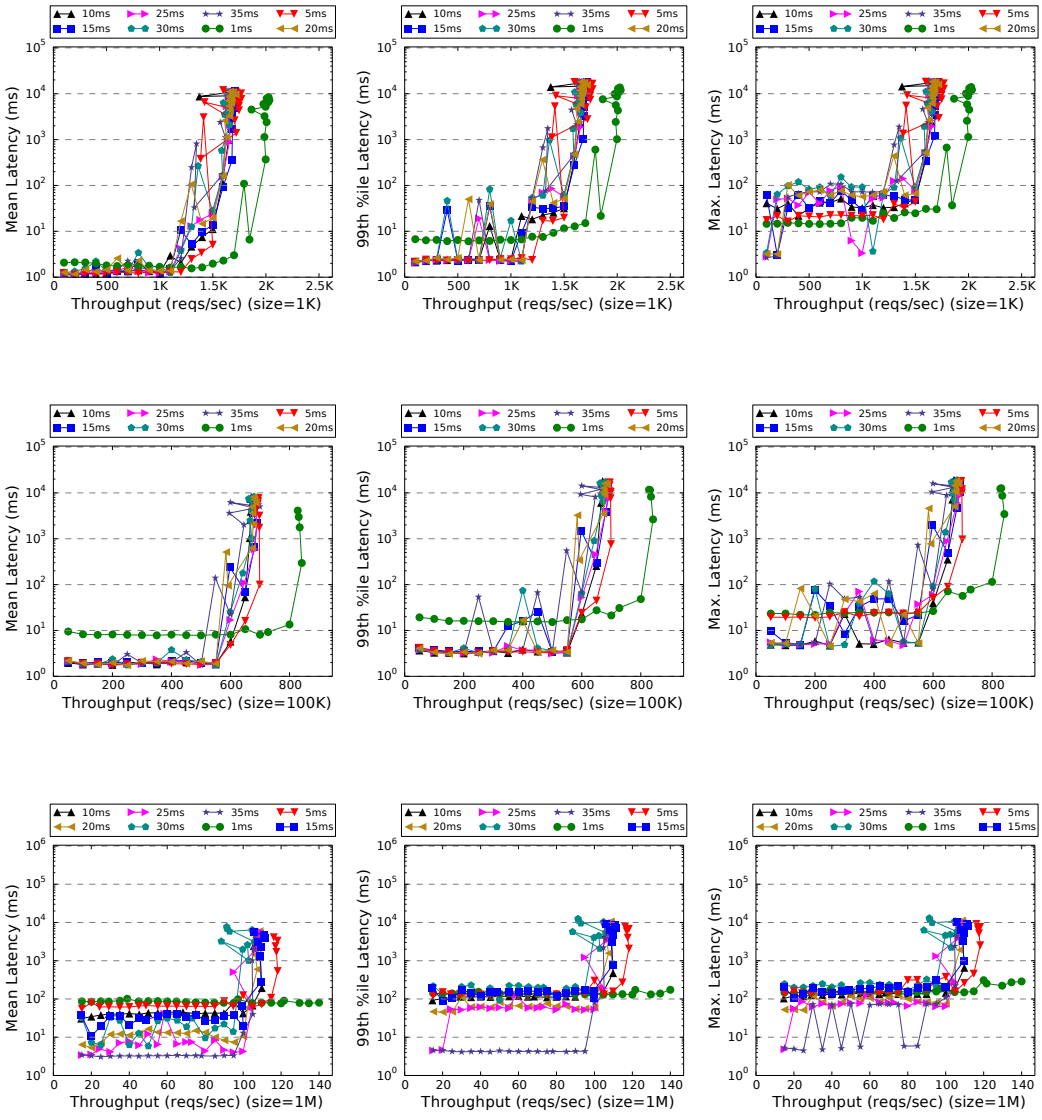


FIGURE A.1: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under Credit, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with varying scheduling latencies and throughput.

A.1.2 Cache-Intensive Background Workload

A.1.2.1 Capped Scenario

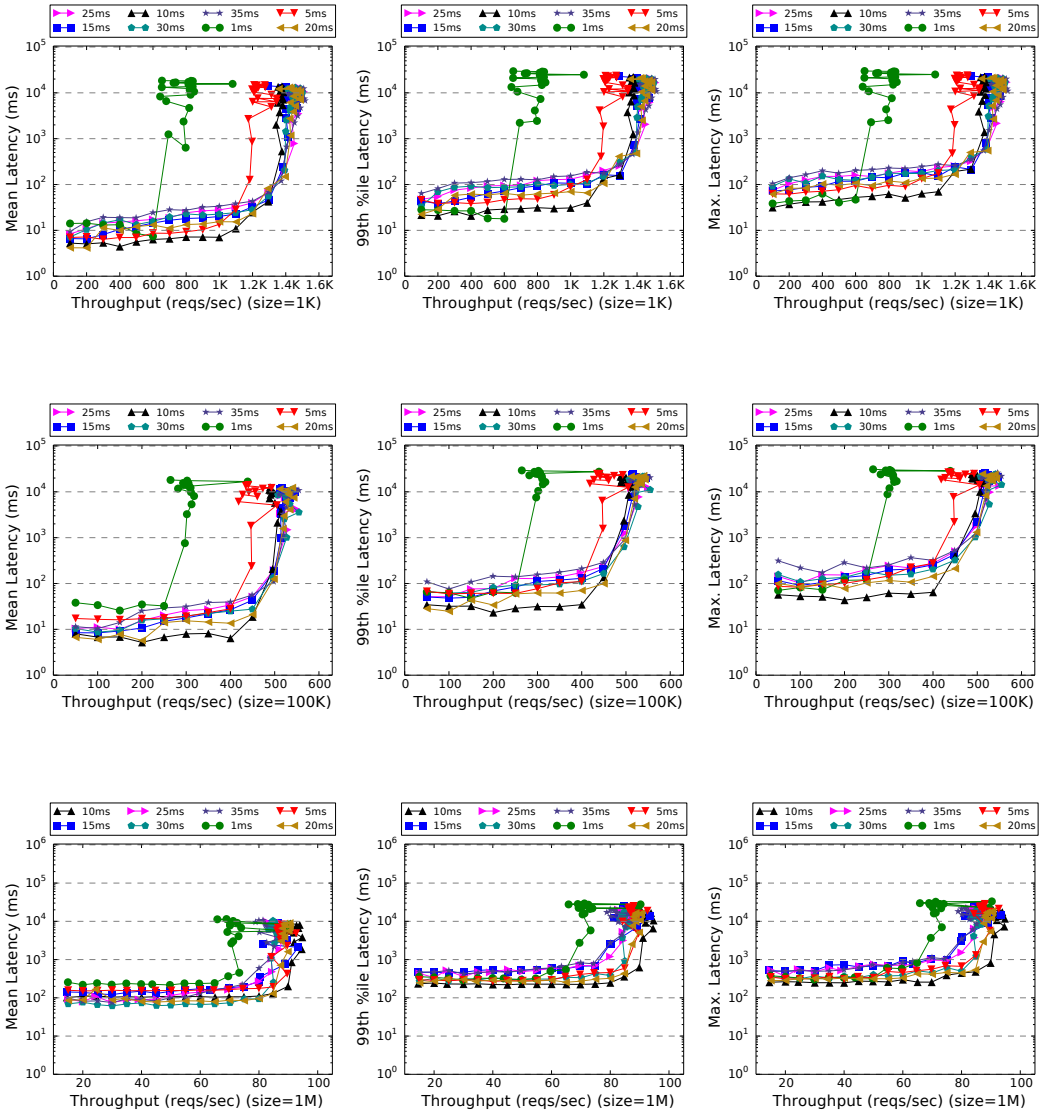


FIGURE A.2: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under Credit, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying scheduling latencies and throughput.

A.1.2.2 Uncapped Scenario

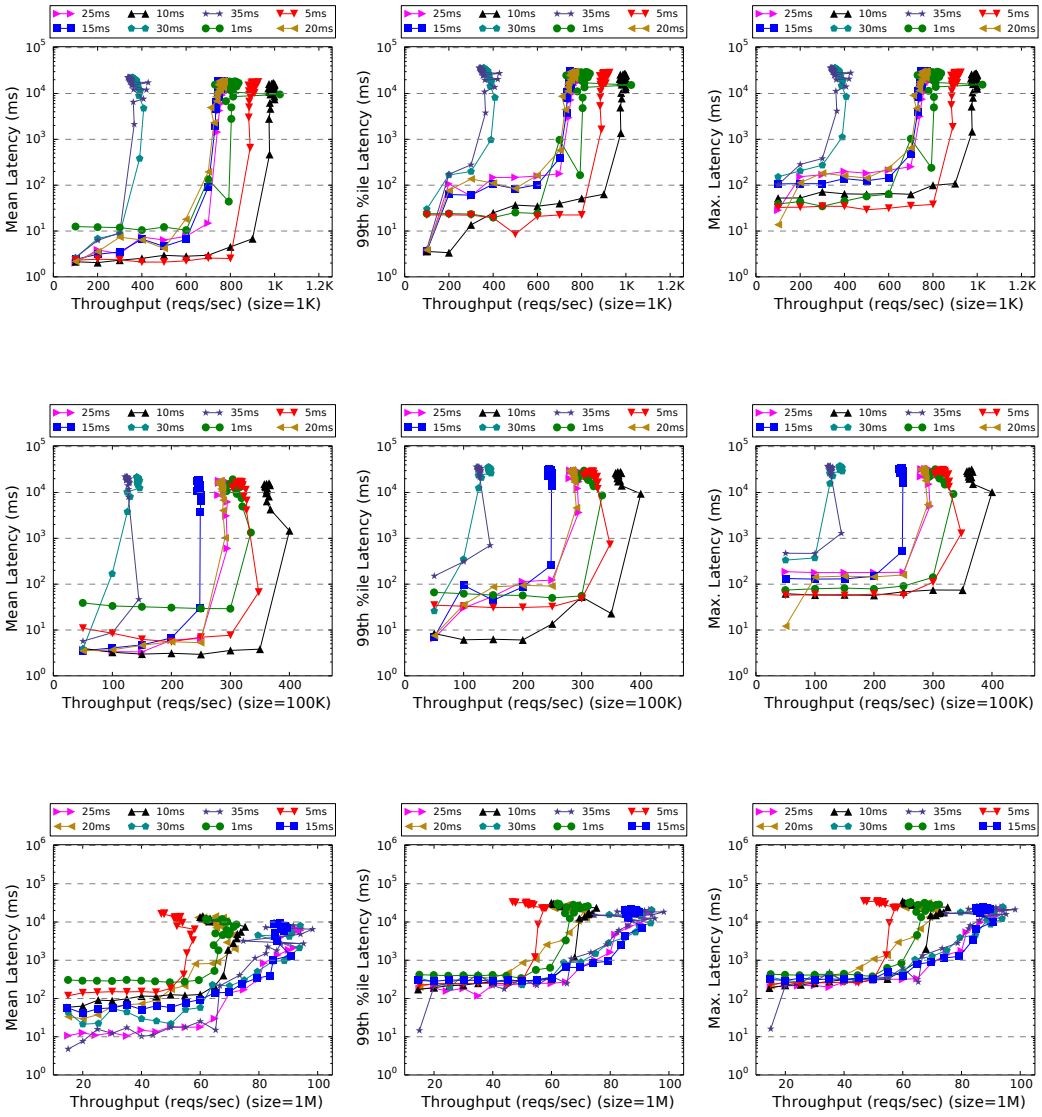


FIGURE A.3: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for an uncapped scenario under Credit, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying scheduling latencies and throughput.

A.1.3 I/O-Intensive Background Workload

A.1.3.1 Capped Scenario

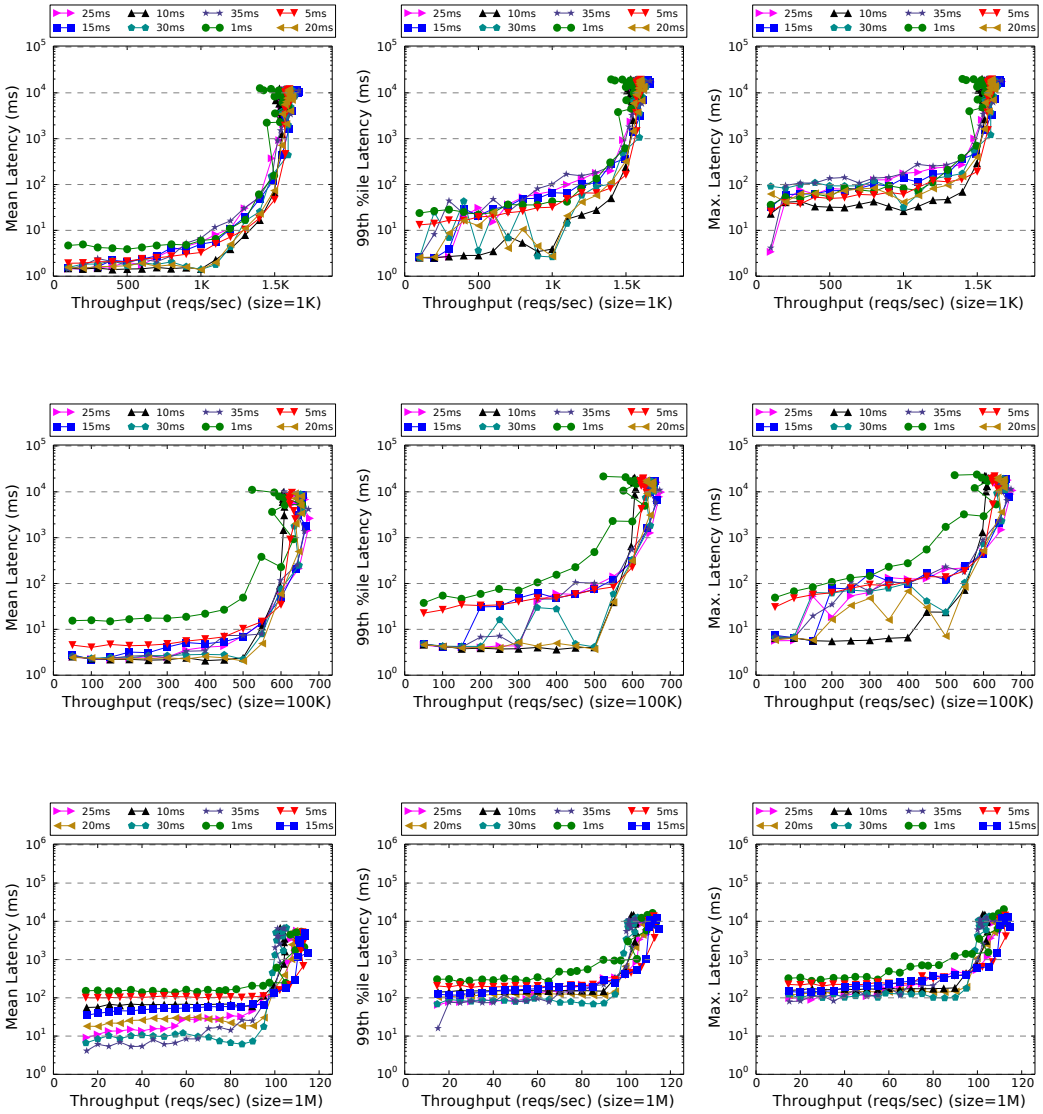


FIGURE A.4: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under Credit, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying scheduling latencies and throughput.

A.1.3.2 Uncapped Scenario

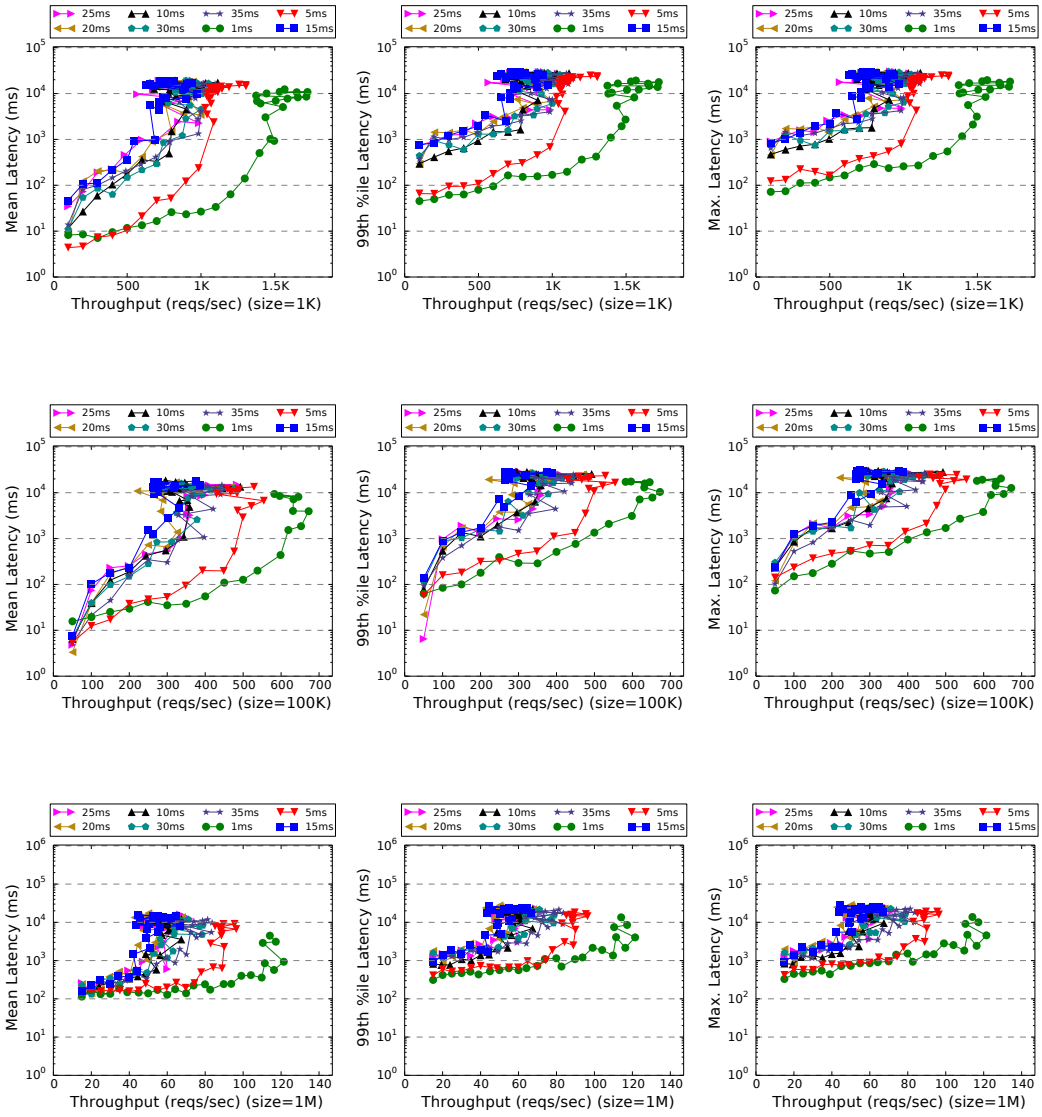


FIGURE A.5: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for an uncapped scenario under Credit, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying scheduling latencies and throughput.

A.2 Tableau Scheduler

A.2.1 Idle Background Workload

A.2.1.1 Capped Scenario

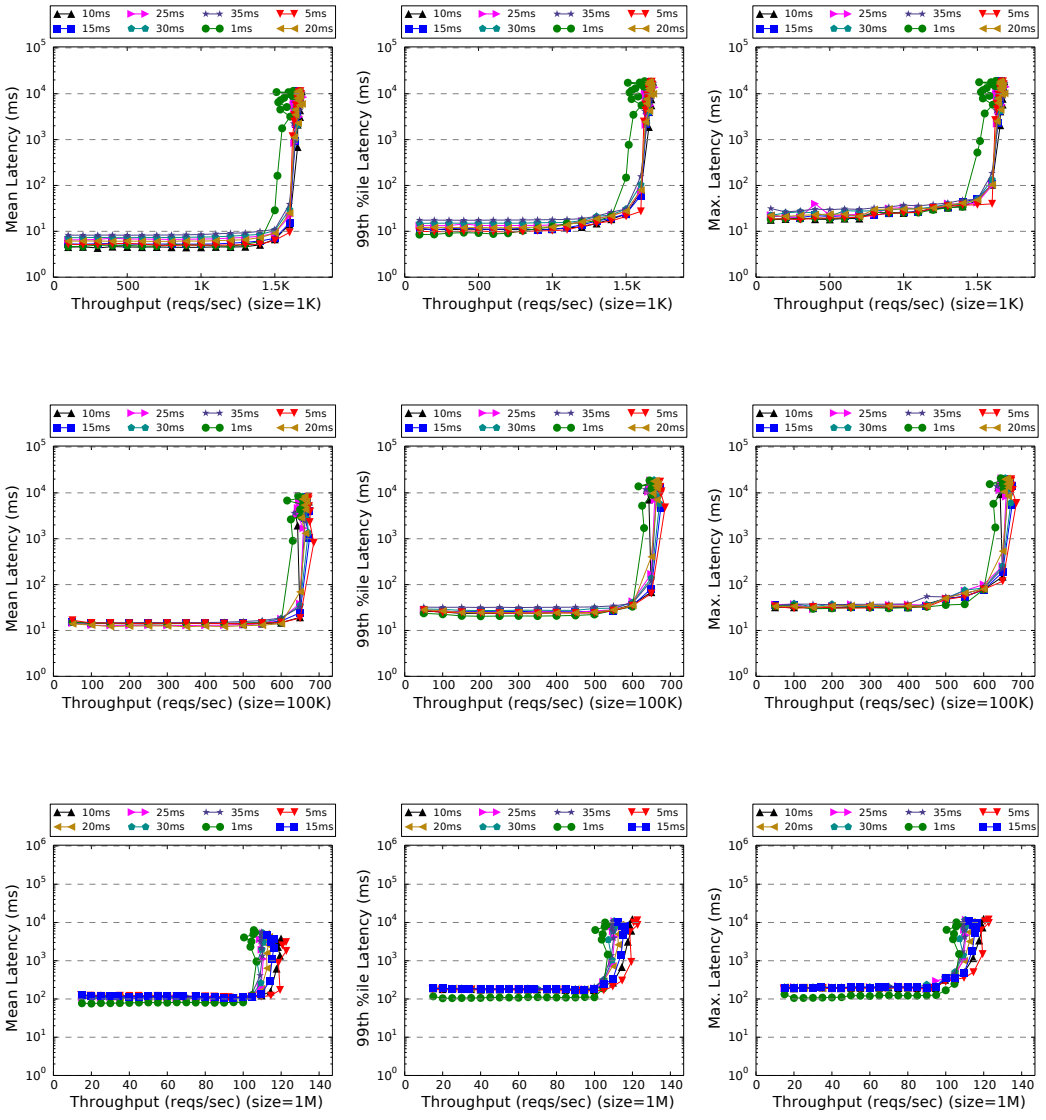


FIGURE A.6: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under Tableau, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with varying scheduling latencies and throughput.

A.2.2 Cache-Intensive Background Workload

A.2.2.1 Capped Scenario

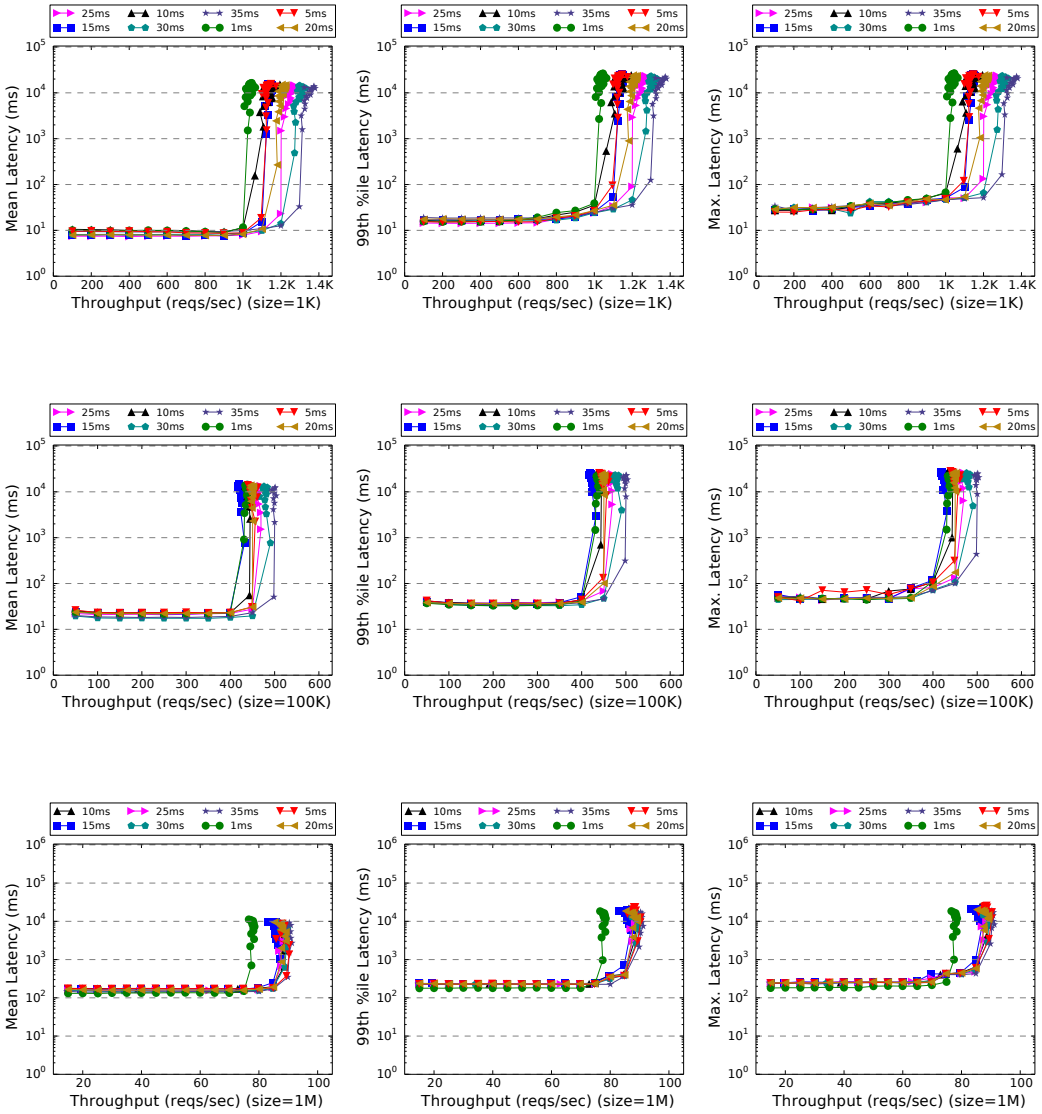


FIGURE A.7: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under Tableau, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying scheduling latencies and throughput.

A.2.2.2 Uncapped Scenario

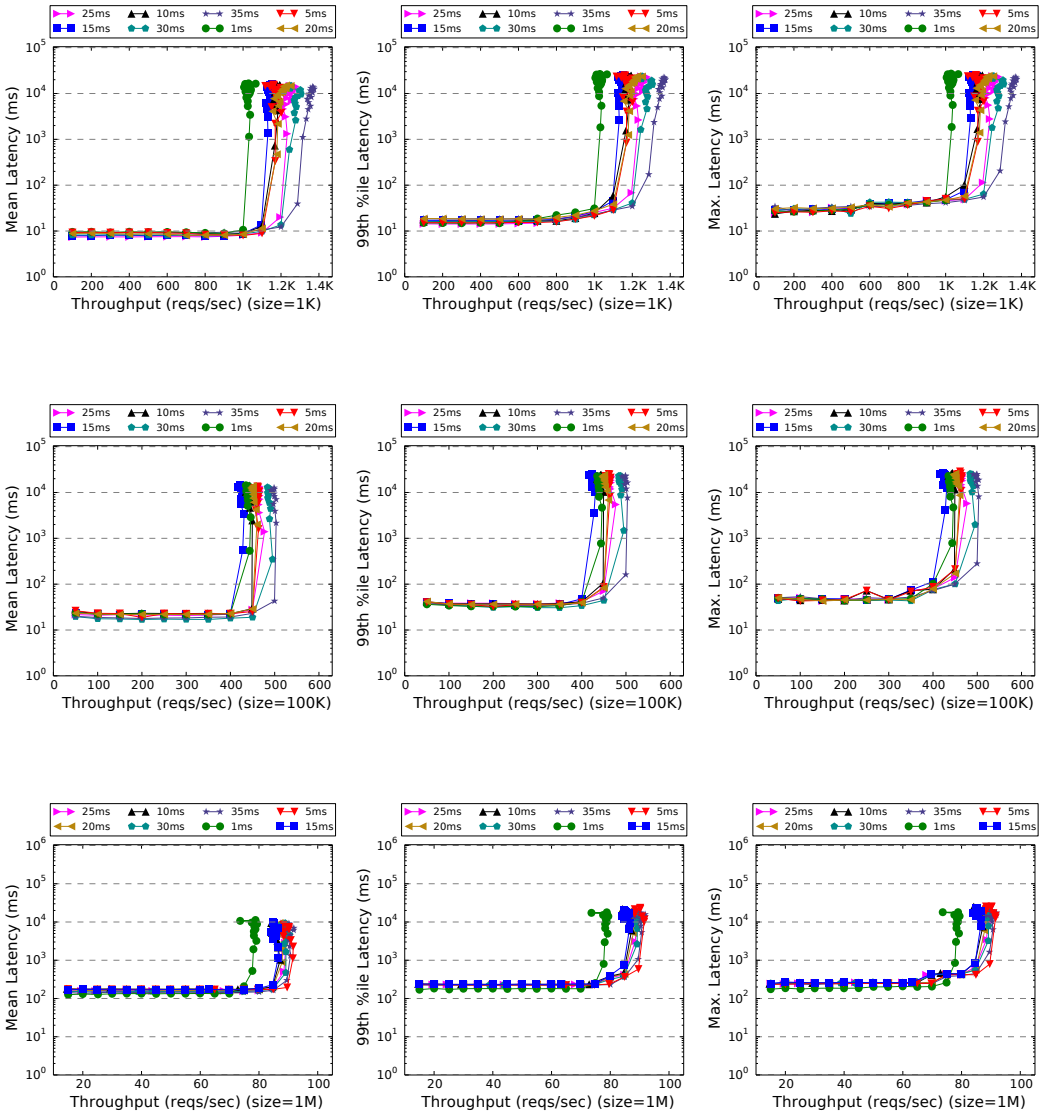


FIGURE A.8: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for an uncapped scenario under Tableau, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying scheduling latencies and throughput.

A.2.3 I/O-Intensive Background Workload

A.2.3.1 Capped Scenario

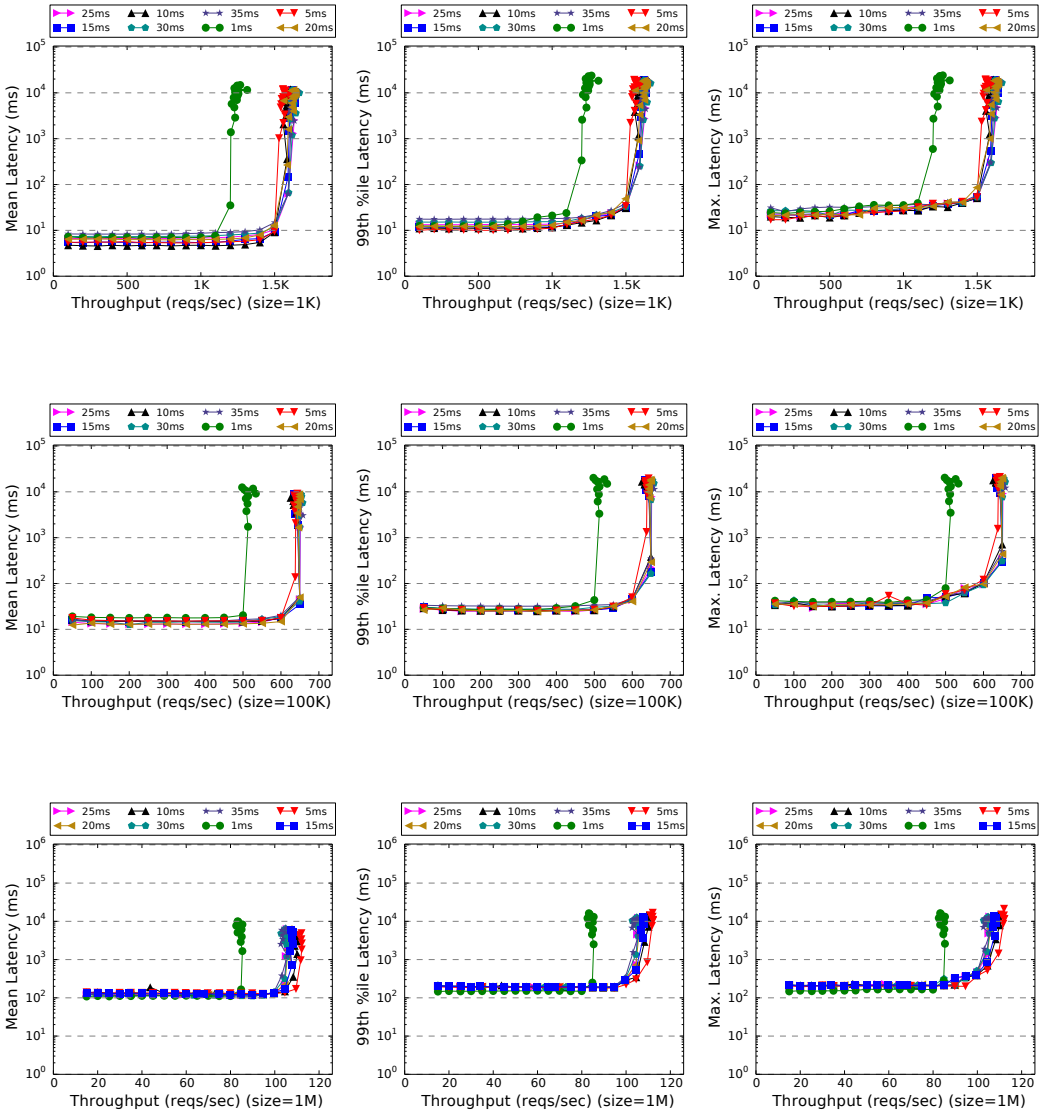


FIGURE A.9: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under Tableau, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying scheduling latencies and throughput.

A.2.3.2 Uncapped Scenario

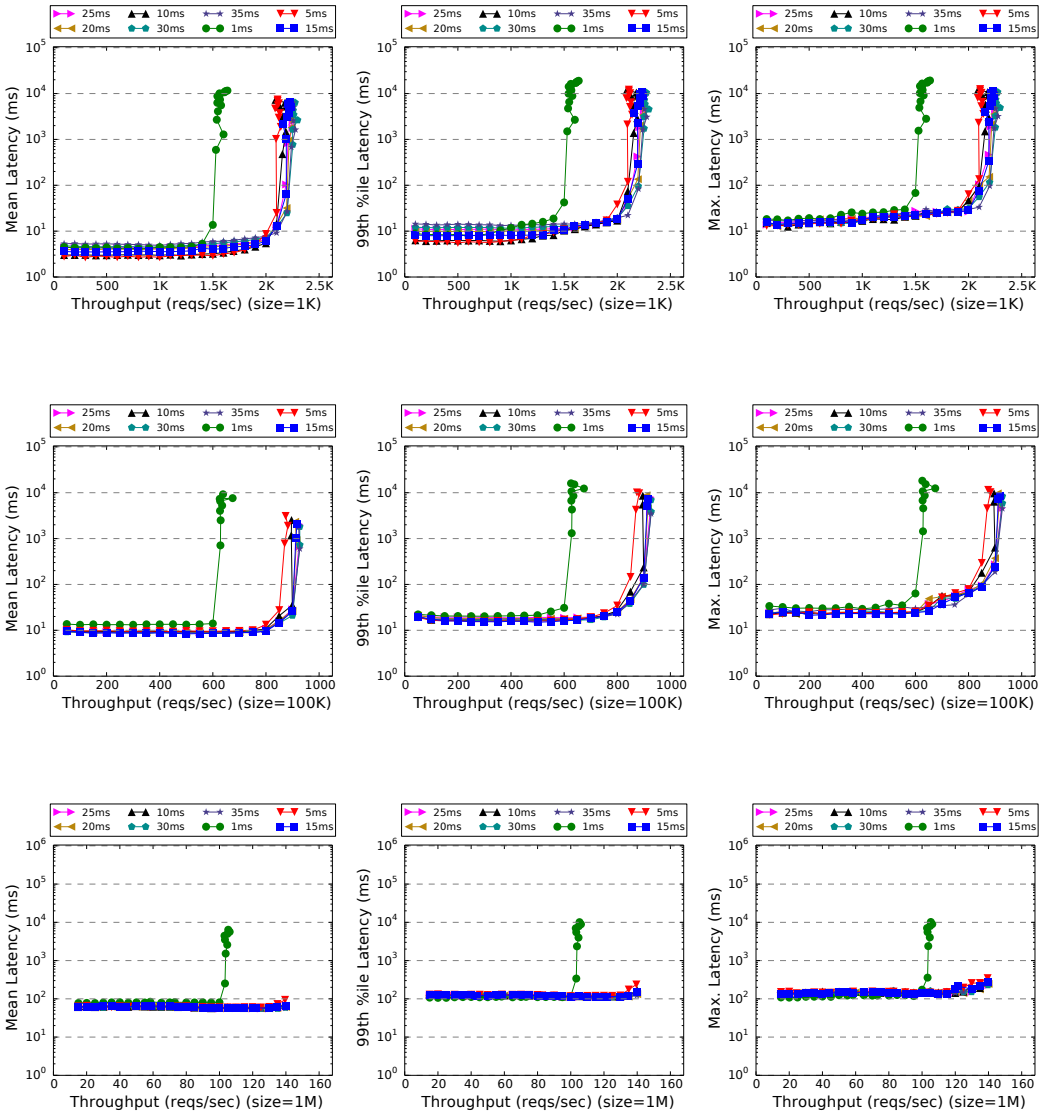


FIGURE A.10: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for an uncapped scenario under Tableau, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying scheduling latencies and throughput.

A.3 RTDS Scheduler

A.3.1 Idle Background Workload

A.3.1.1 Capped Scenario

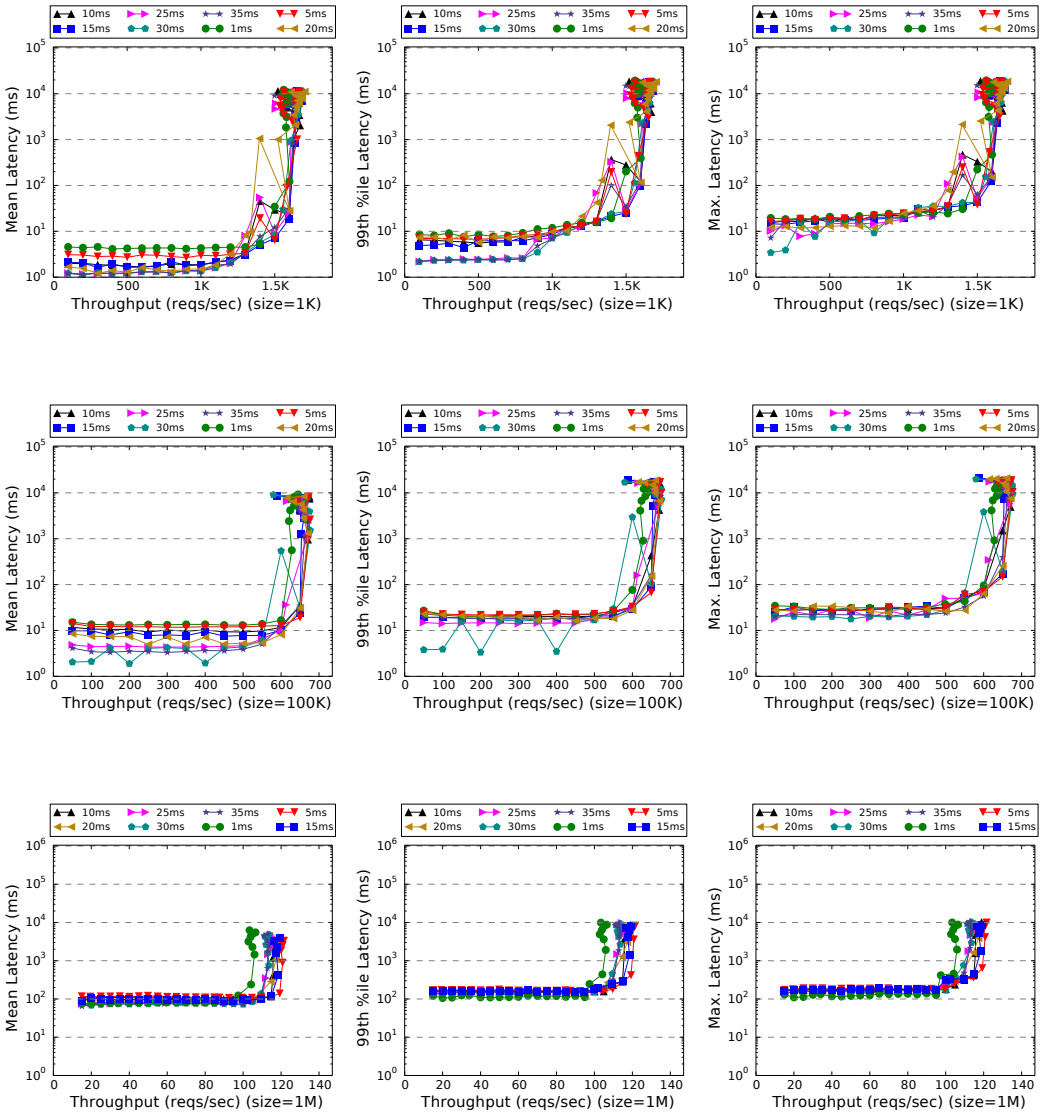


FIGURE A.11: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under RTDS, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with varying scheduling latencies and throughput.

A.3.2 Cache-Intensive Background Workload

A.3.2.1 Capped Scenario

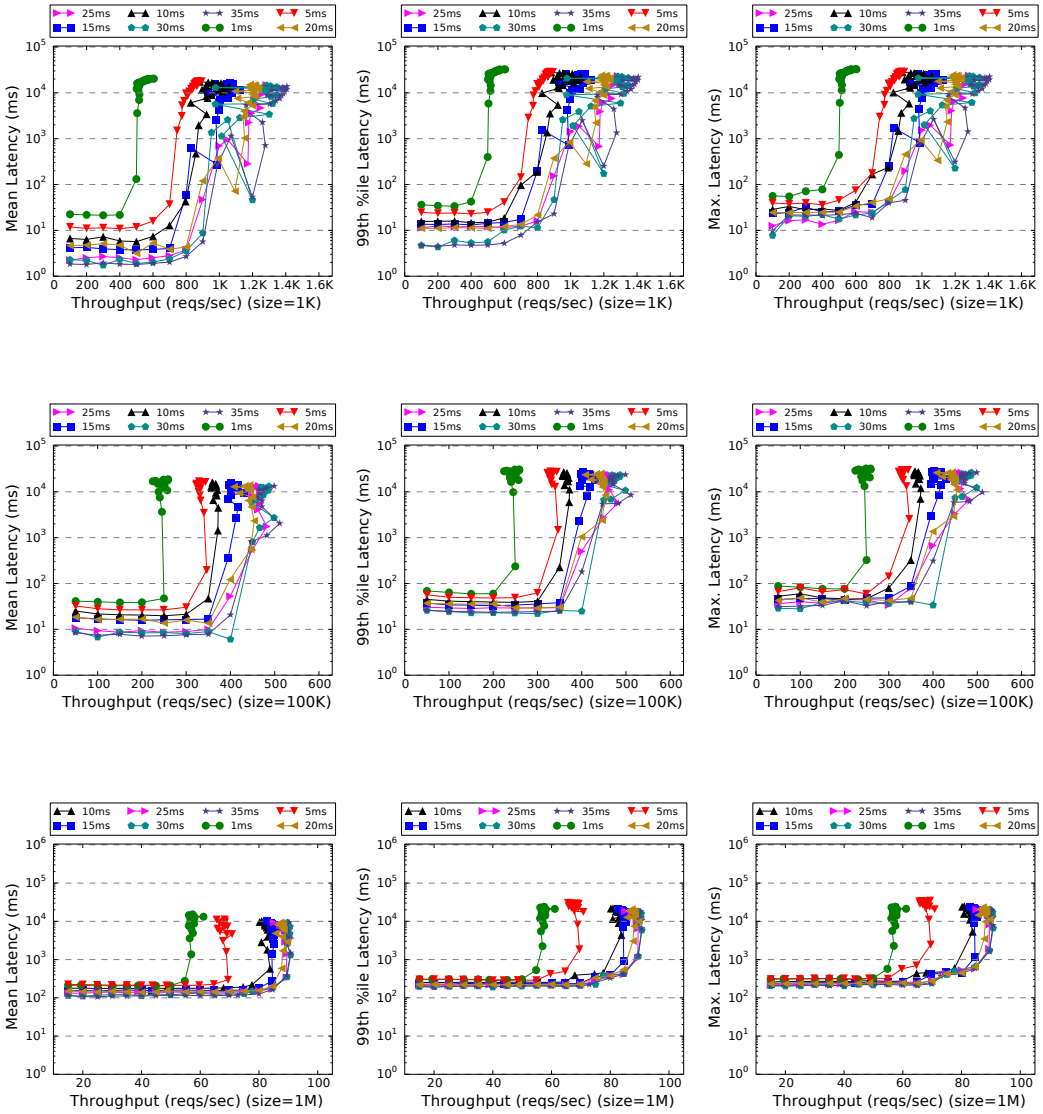


FIGURE A.12: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under RTDS, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying scheduling latencies and throughput.

A.3.3 I/O-Intensive Background Workload

A.3.3.1 Capped Scenario

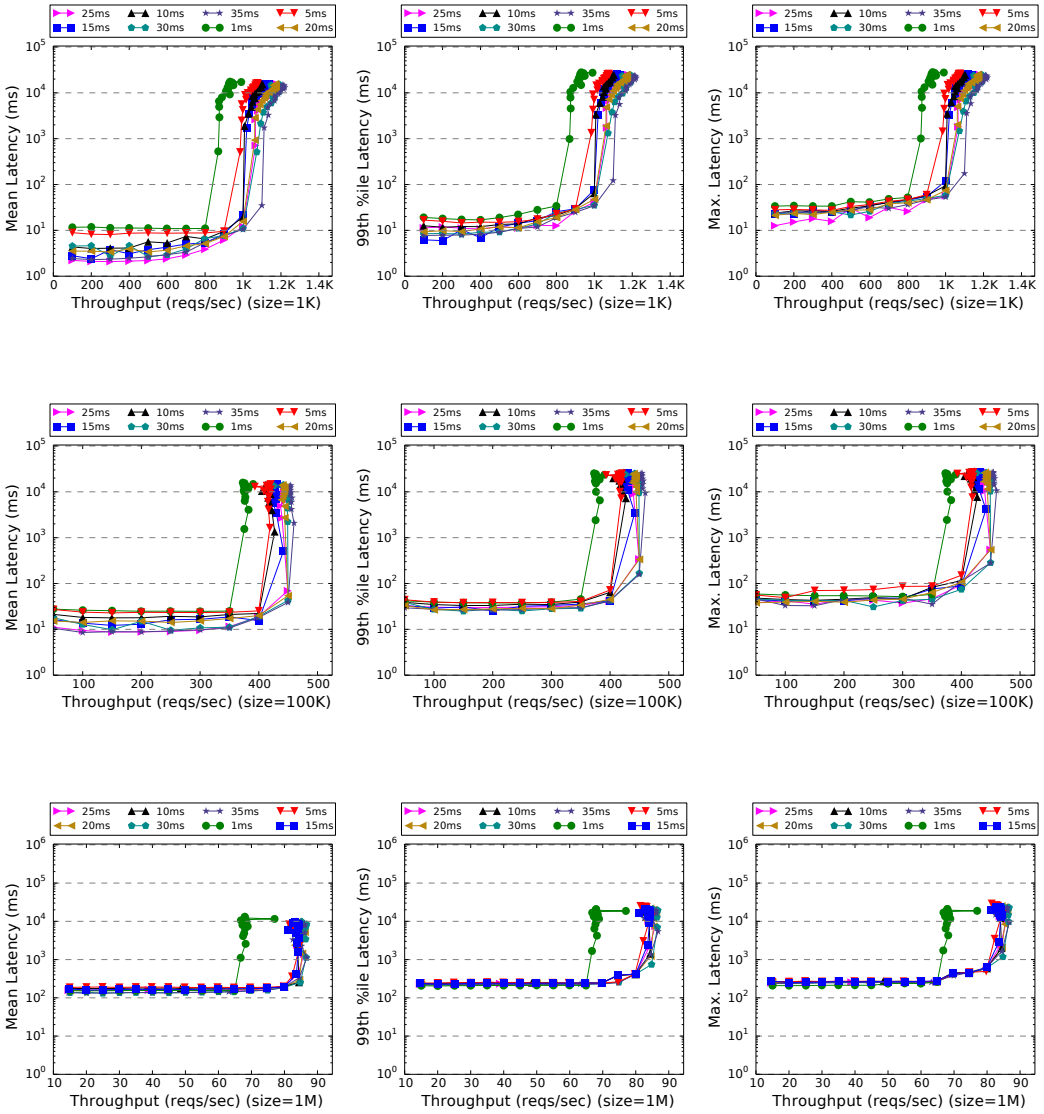


FIGURE A.13: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for a capped scenario under RTDS, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying scheduling latencies and throughput.

Appendix B

Extended Evaluation: Scheduling Latency Across Schedulers

This section compares the performance of the four evaluated Xen schedulers (Credit, Credit2, RTDS, and Tableau) with varying configurations for scheduling latency. The details of the experimental setup are described in detail in section 6.4, and the data presented in this section is simply a different representation of the data presented in Appendix A.

In the scenario with an *idle background workload* and *uncapped VMs*, the performance is similar to the full-core scenario. Therefore these graphs are omitted entirely in this section. To see a comparison of performance of each scheduler under a full-core scenario, please refer to Appendix C.

Further, for the Credit2 scheduler, since there is no way to configure the scheduling, a single set of results are used in all graphs to allow for comparison.

Note that, in general, one overall trend that appears across schedulers is that a 1 ms timeslice or scheduling latency results in the lowest performance, regardless of background workload or capped or uncapped scenario. This is simply due to the large number of interrupts generated by the scheduler causing a reduction in throughput due to the additional overheads.

B.1 1ms Scheduling Latency

B.1.1 Idle Background Workload

B.1.1.1 Capped Scenario

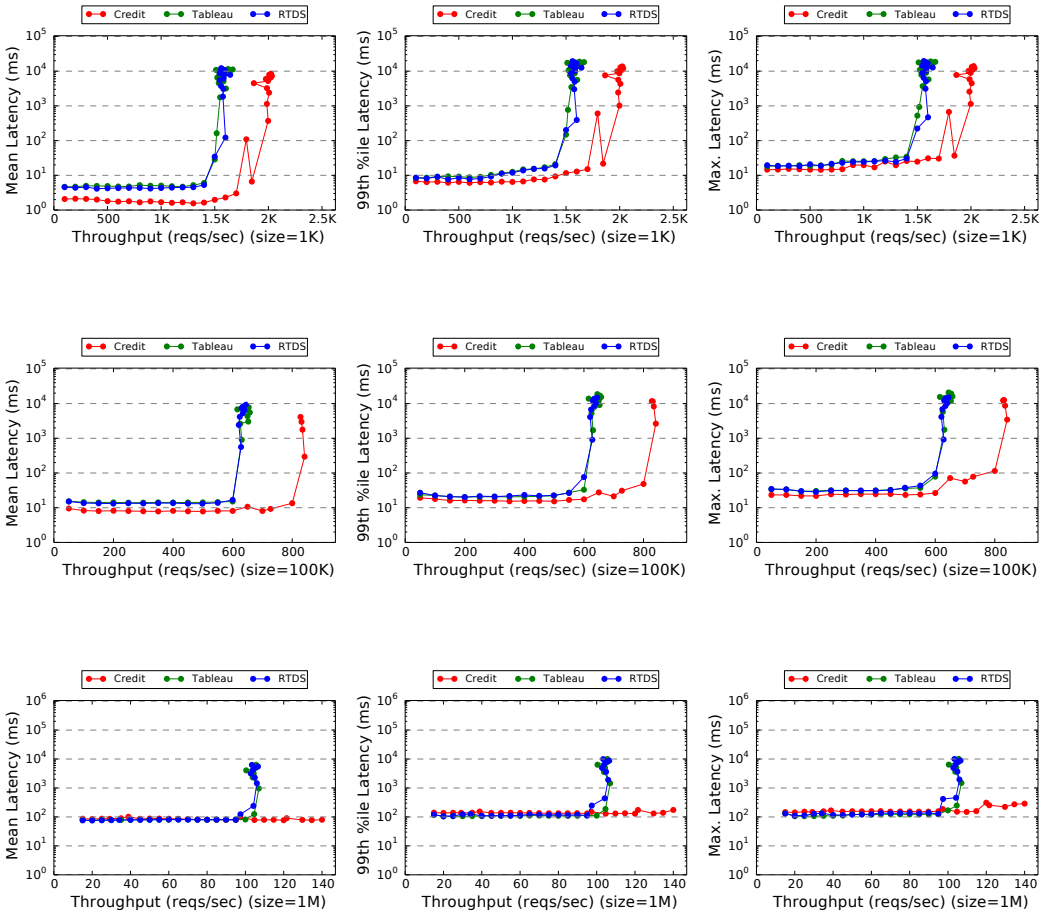


FIGURE B.1: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 1ms timeslice (or scheduling latency) and varying throughput.

B.1.2 Cache-Intensive Background Workload

B.1.2.1 Capped Scenario

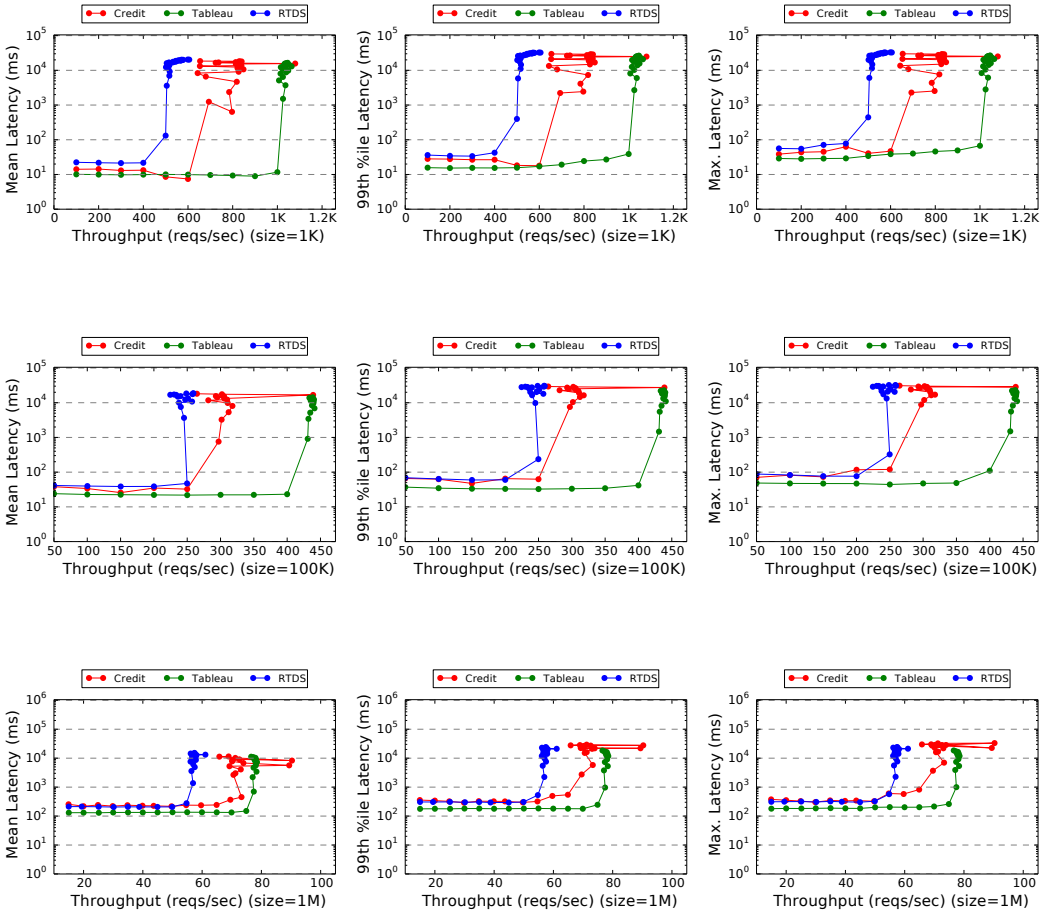


FIGURE B.2: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 1ms timeslice (or scheduling latency) and varying throughput.

B.1.2.2 Uncapped Scenario

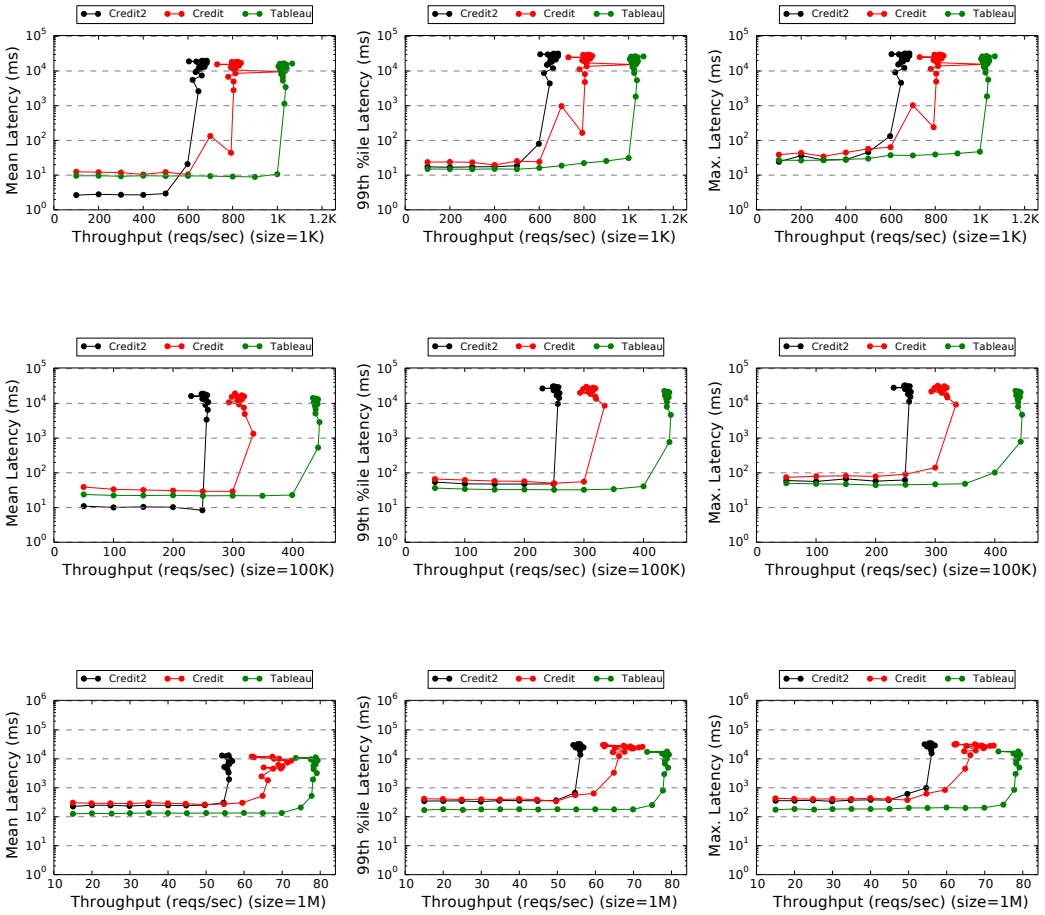


FIGURE B.3: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 1ms timeslice (or scheduling latency) and varying throughput.

B.1.3 I/O-Intensive Background Workload

B.1.3.1 Capped Scenario

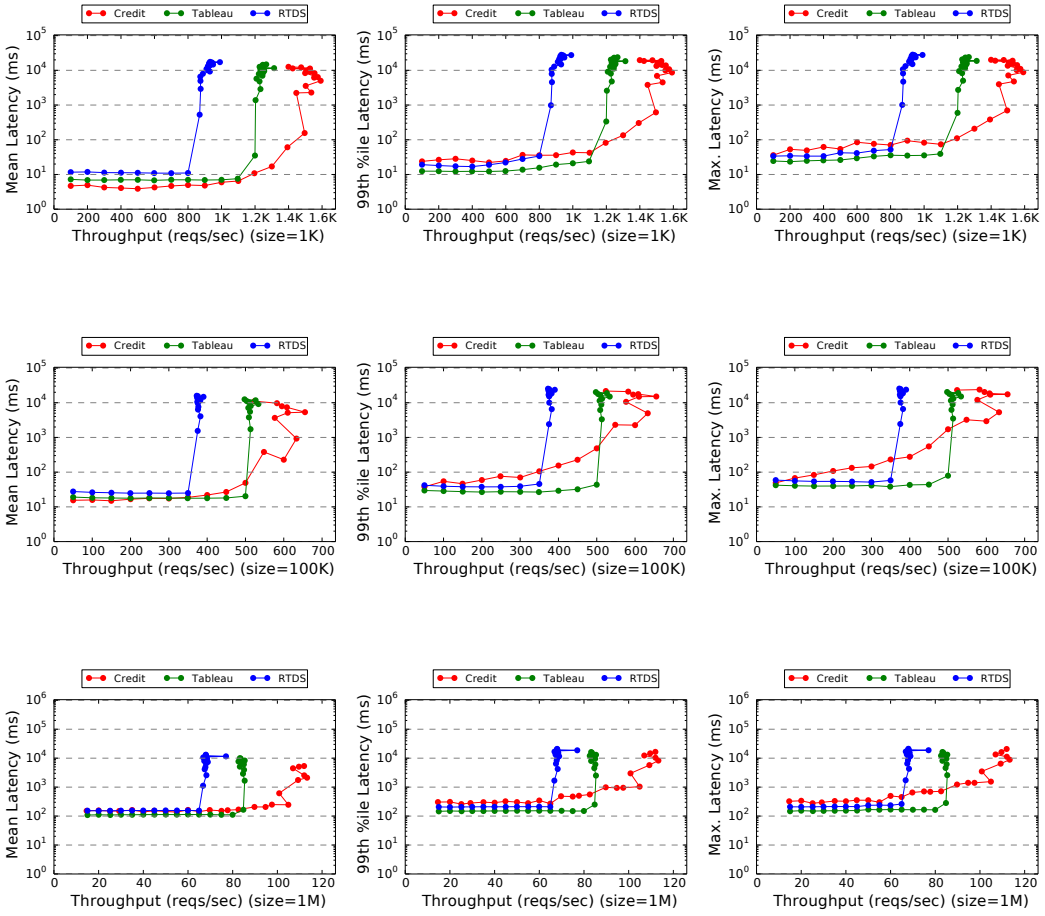


FIGURE B.4: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 1ms timeslice (or scheduling latency) and varying throughput.

B.1.3.2 Uncapped Scenario

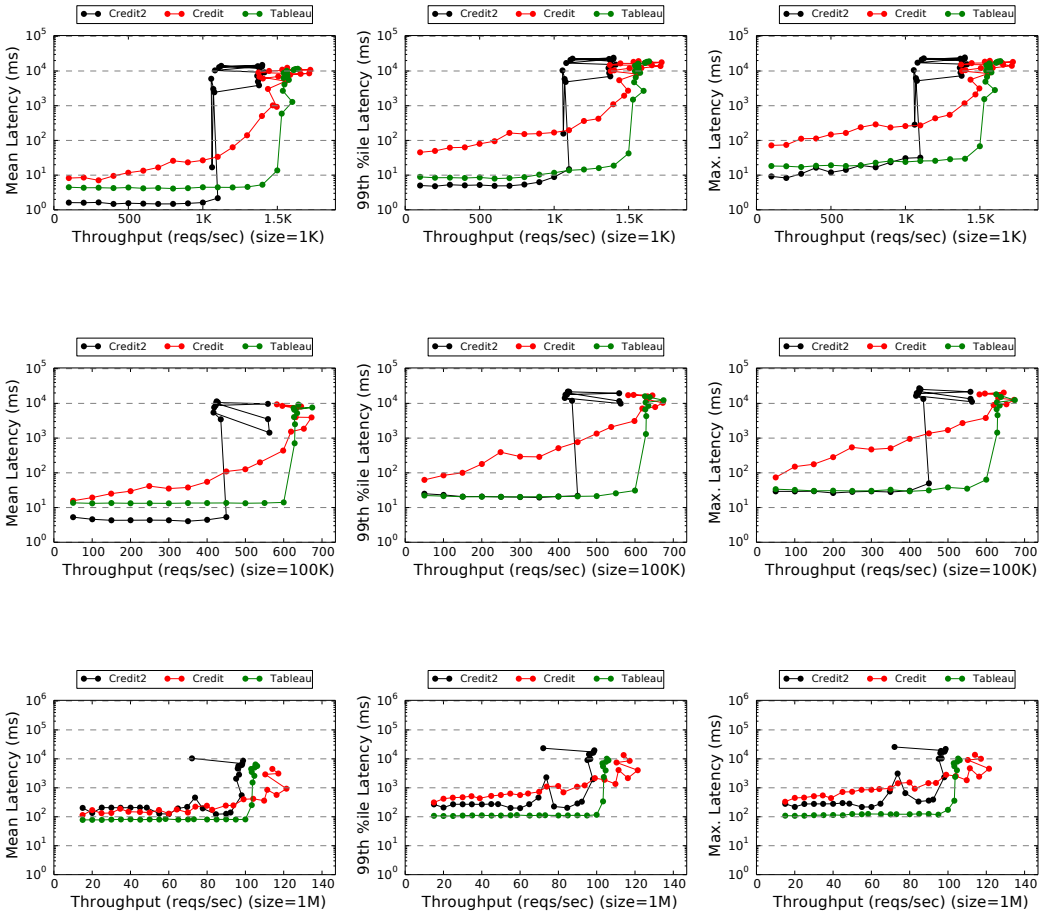


FIGURE B.5: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 1ms timeslice (or scheduling latency) and varying throughput.

B.2 5ms Scheduling Latency

B.2.1 Idle Background Workload

B.2.1.1 Capped Scenario

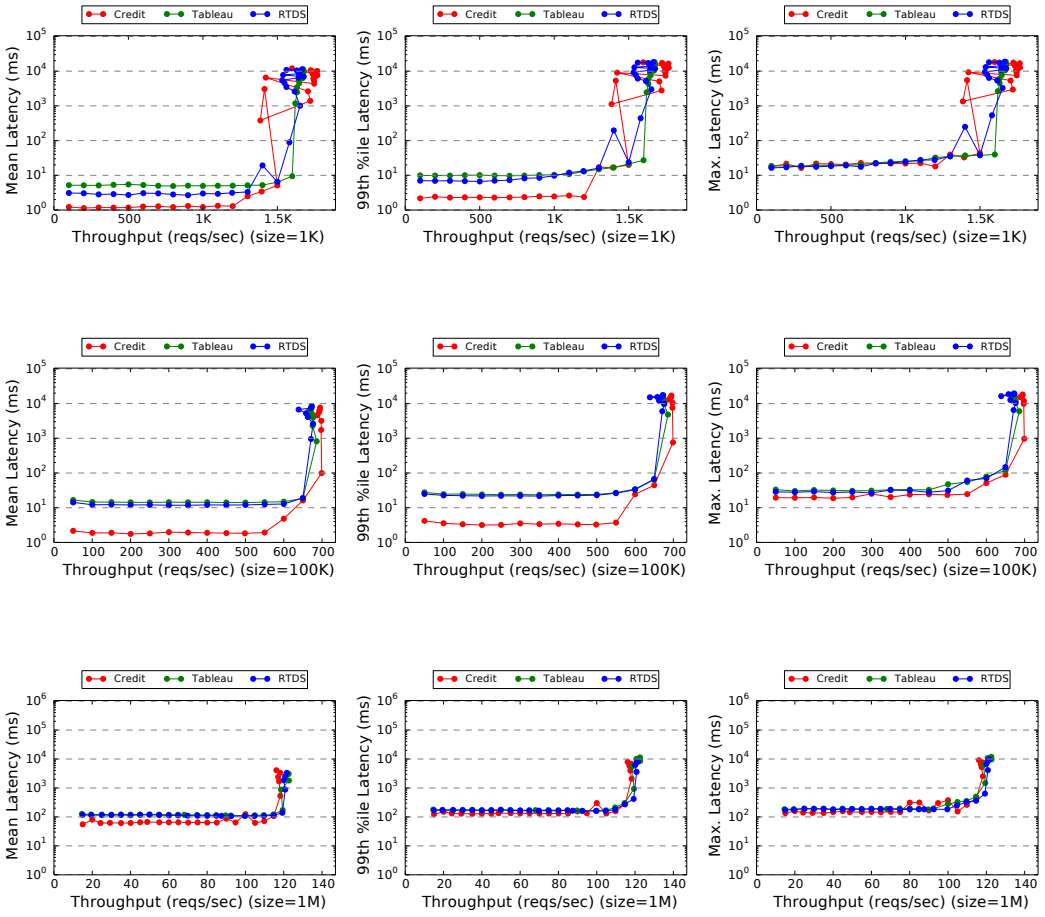


FIGURE B.6: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 5ms timeslice (or scheduling latency) and varying throughput.

B.2.2 Cache-Intensive Background Workload

B.2.2.1 Capped Scenario

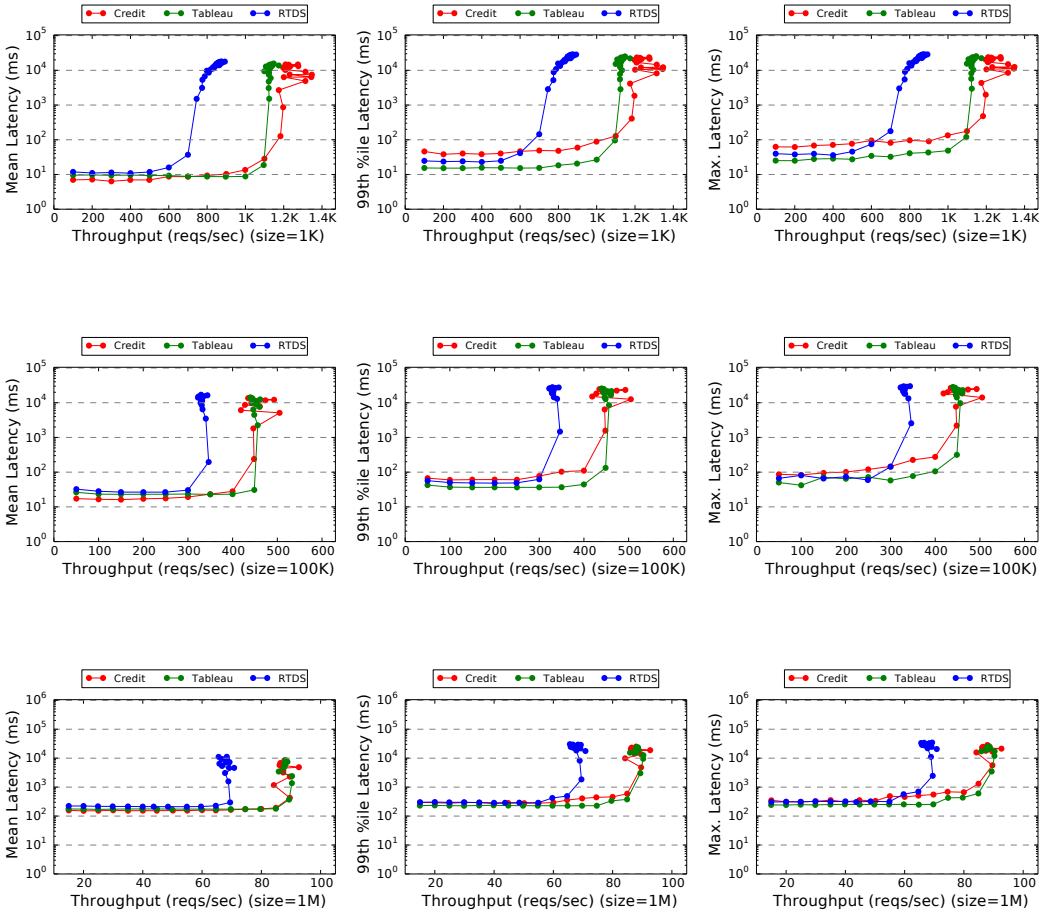


FIGURE B.7: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 5ms timeslice (or scheduling latency) and varying throughput.

B.2.2.2 Uncapped Scenario

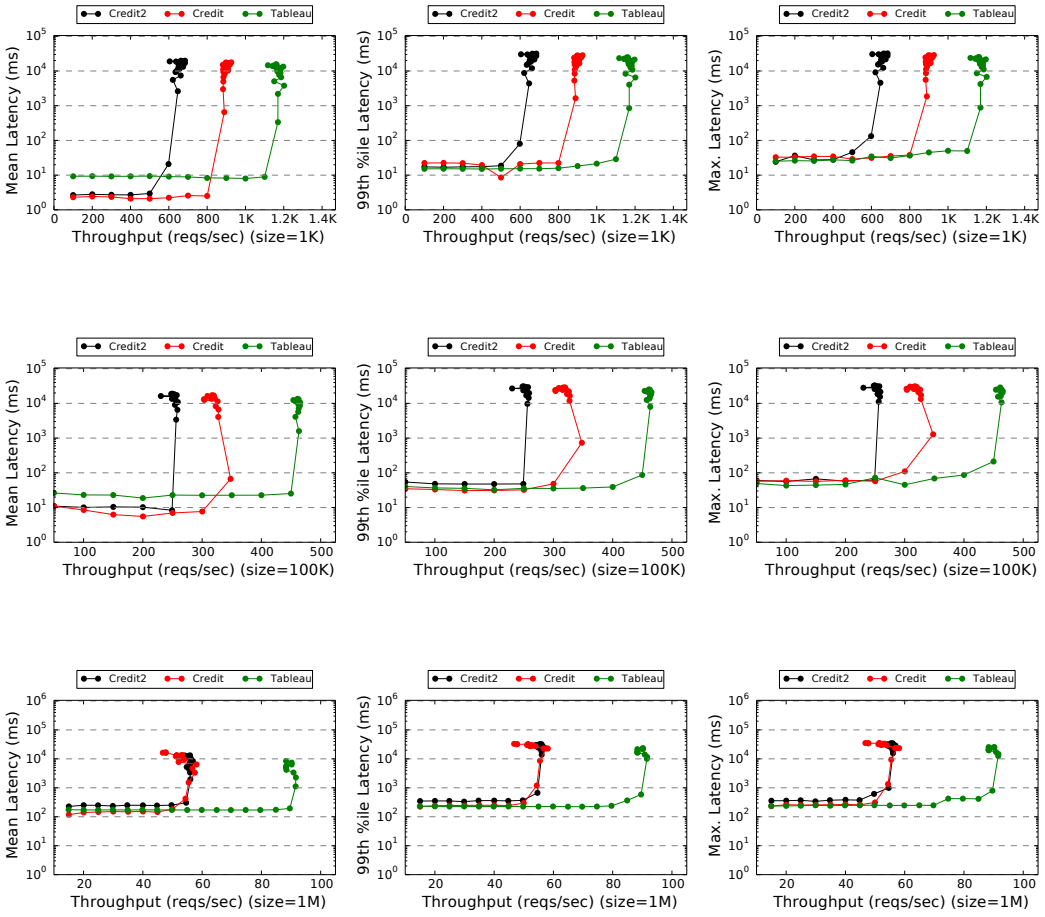


FIGURE B.8: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 5ms timeslice (or scheduling latency) and varying throughput.

B.2.3 I/O-Intensive Background Workload

B.2.3.1 Capped Scenario

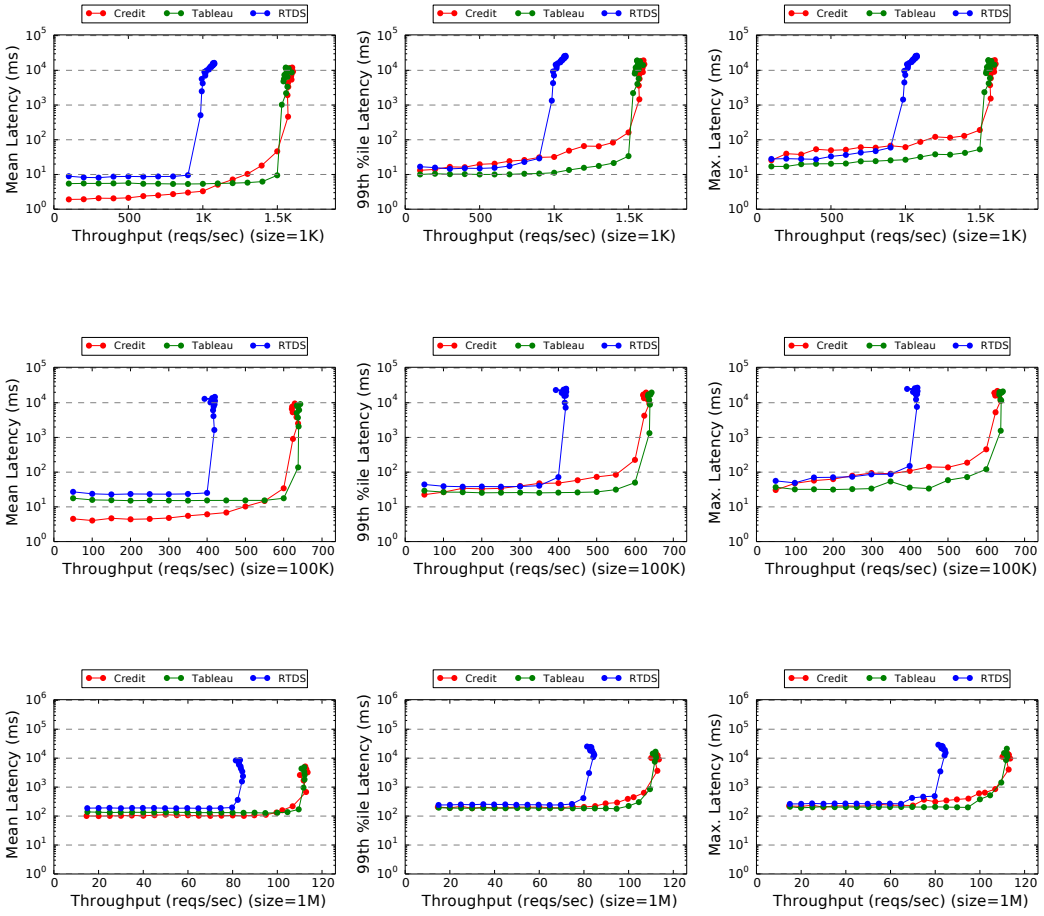


FIGURE B.9: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 5ms timeslice (or scheduling latency) and varying throughput.

B.2.3.2 Uncapped Scenario

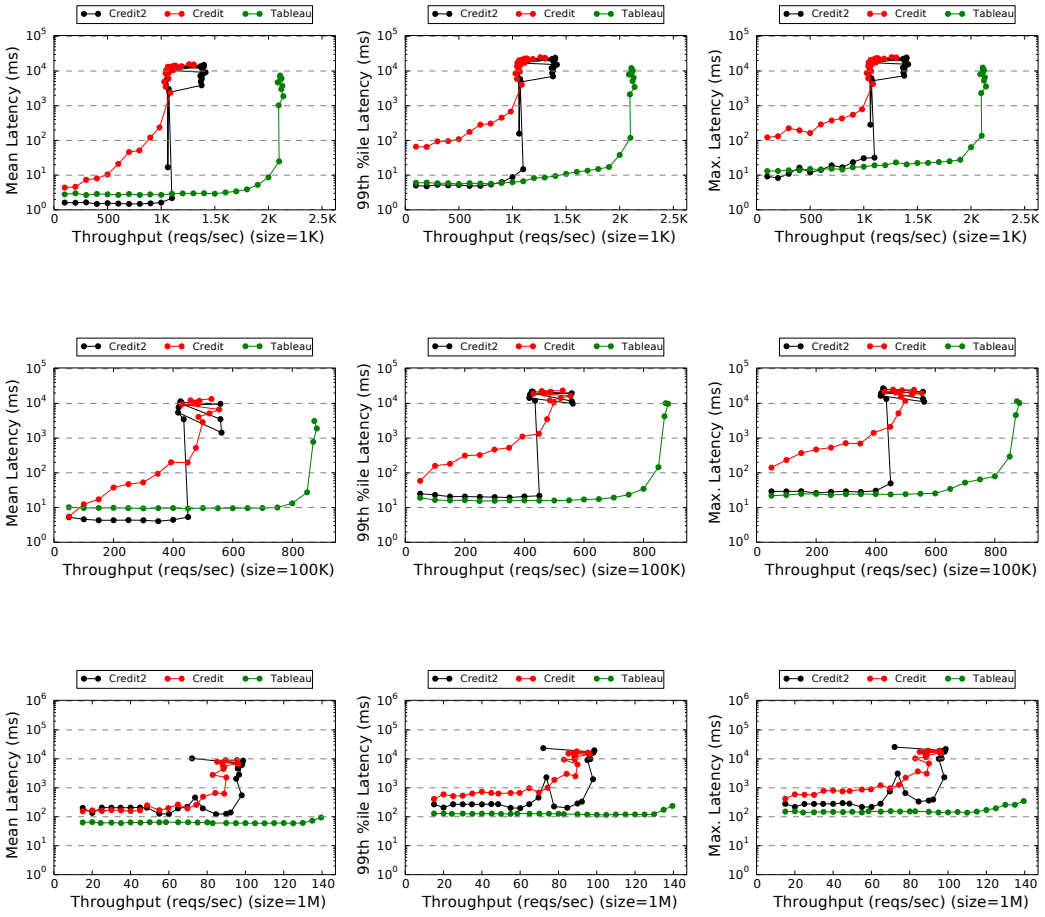


FIGURE B.10: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 5ms timeslice (or scheduling latency) and varying throughput.

B.3 10ms Scheduling Latency

B.3.1 Idle Background Workload

B.3.1.1 Capped Scenario

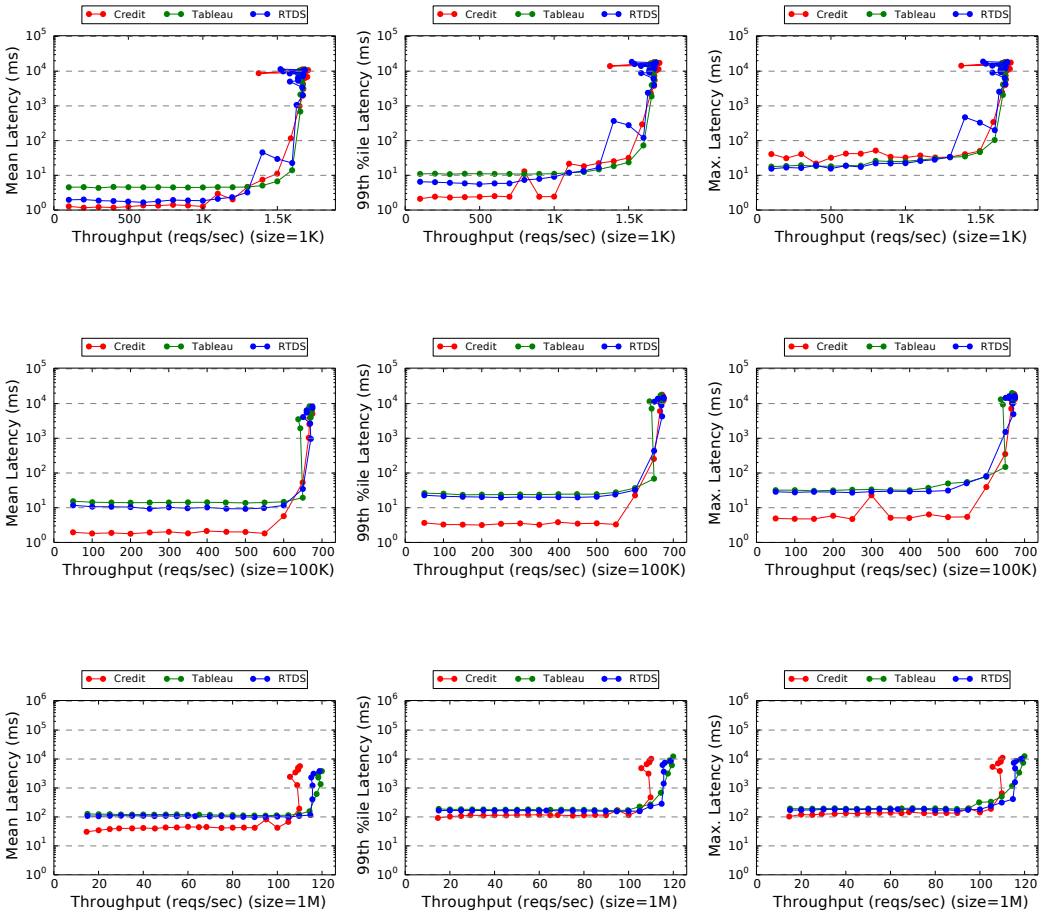


FIGURE B.11: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 10ms timeslice (or scheduling latency) and varying throughput.

B.3.2 Cache-Intensive Background Workload

B.3.2.1 Capped Scenario

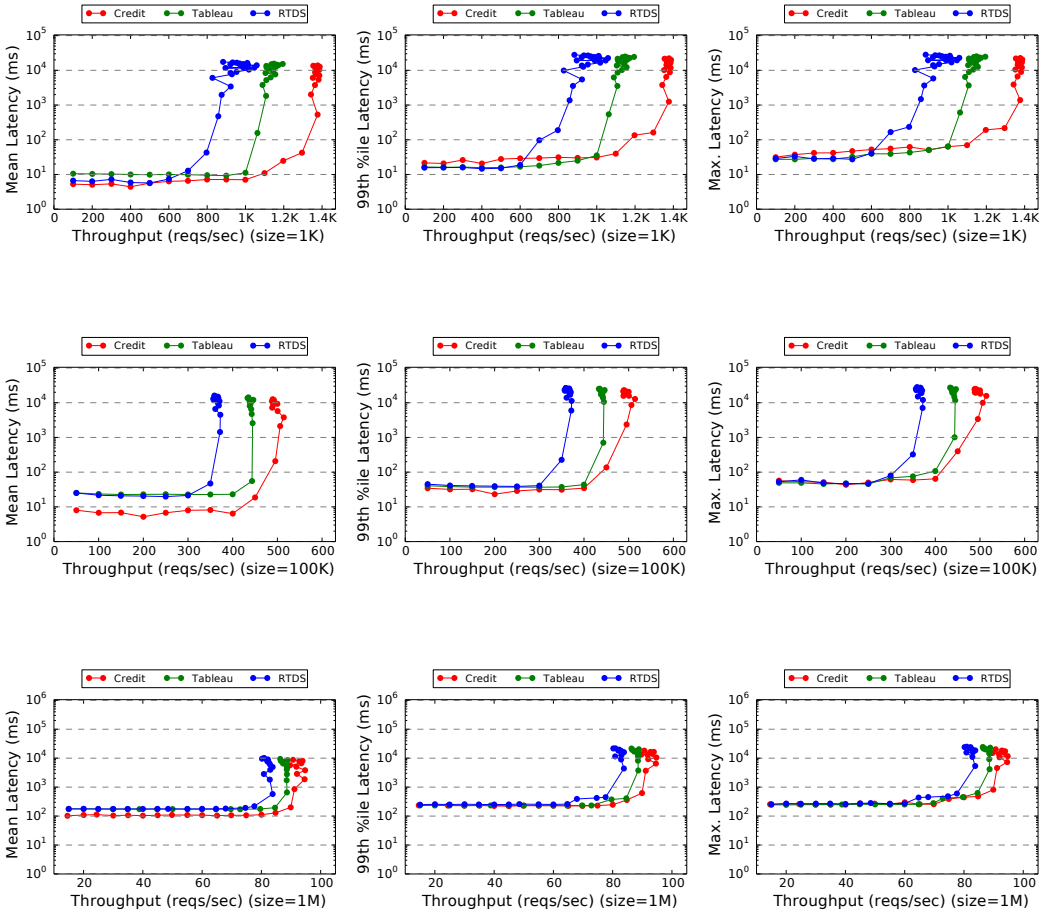


FIGURE B.12: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 10ms timeslice (or scheduling latency) and varying throughput.

B.3.2.2 Uncapped Scenario

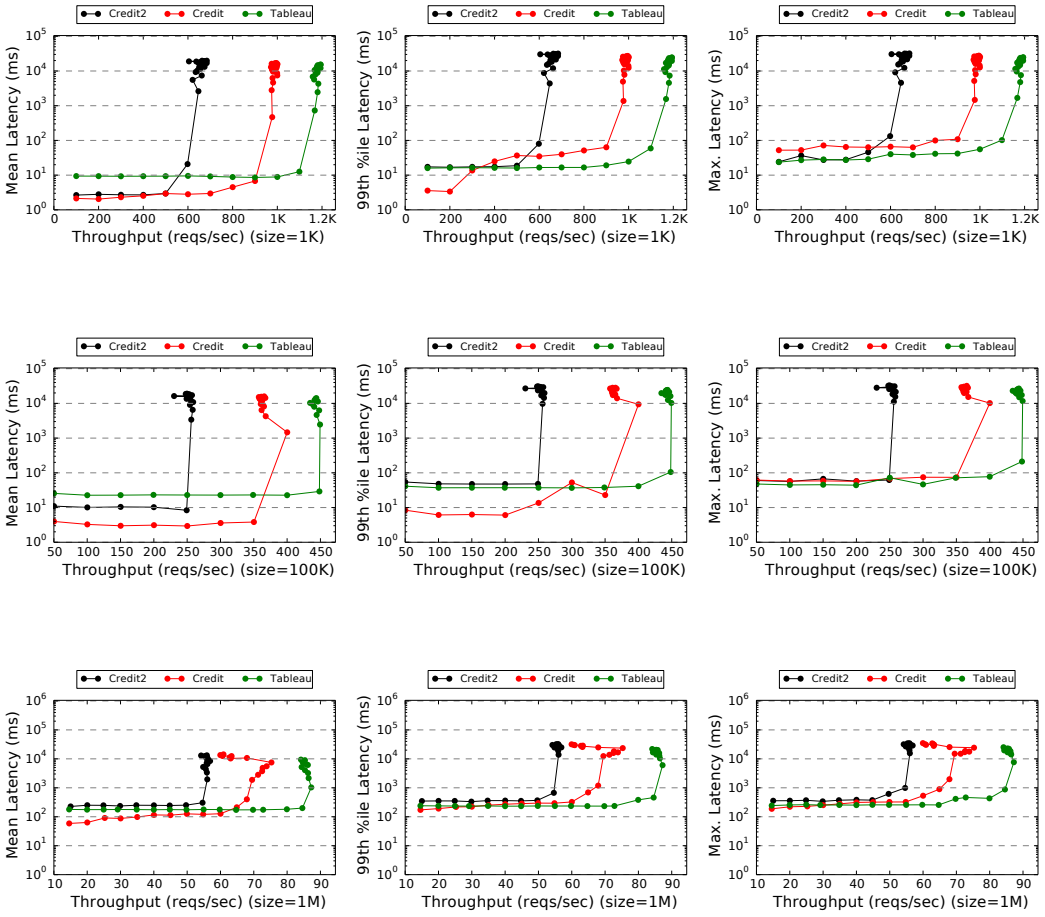


FIGURE B.13: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 10ms timeslice (or scheduling latency) and varying throughput.

B.3.3 I/O-Intensive Background Workload

B.3.3.1 Capped Scenario

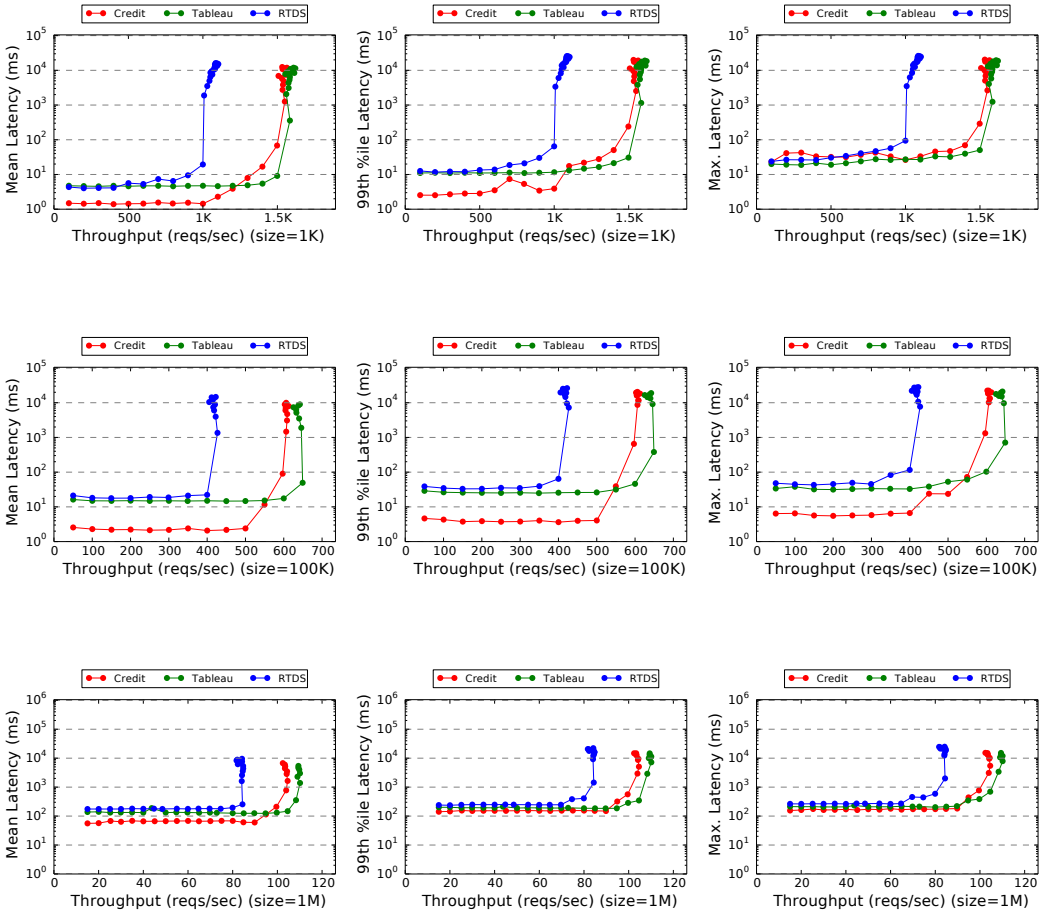


FIGURE B.14: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 10ms timeslice (or scheduling latency) and varying throughput.

B.3.3.2 Uncapped Scenario

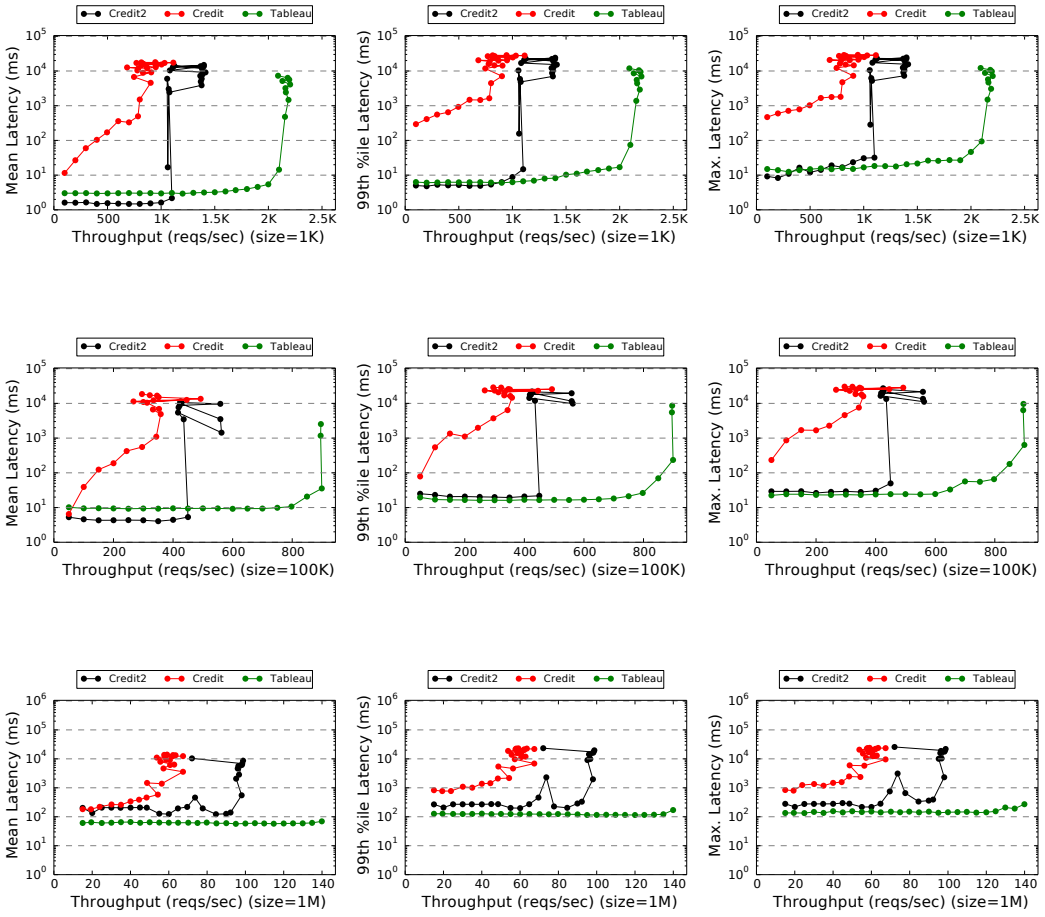


FIGURE B.15: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 10ms timeslice (or scheduling latency) and varying throughput.

B.4 15ms Scheduling Latency

B.4.1 Idle Background Workload

B.4.1.1 Capped Scenario

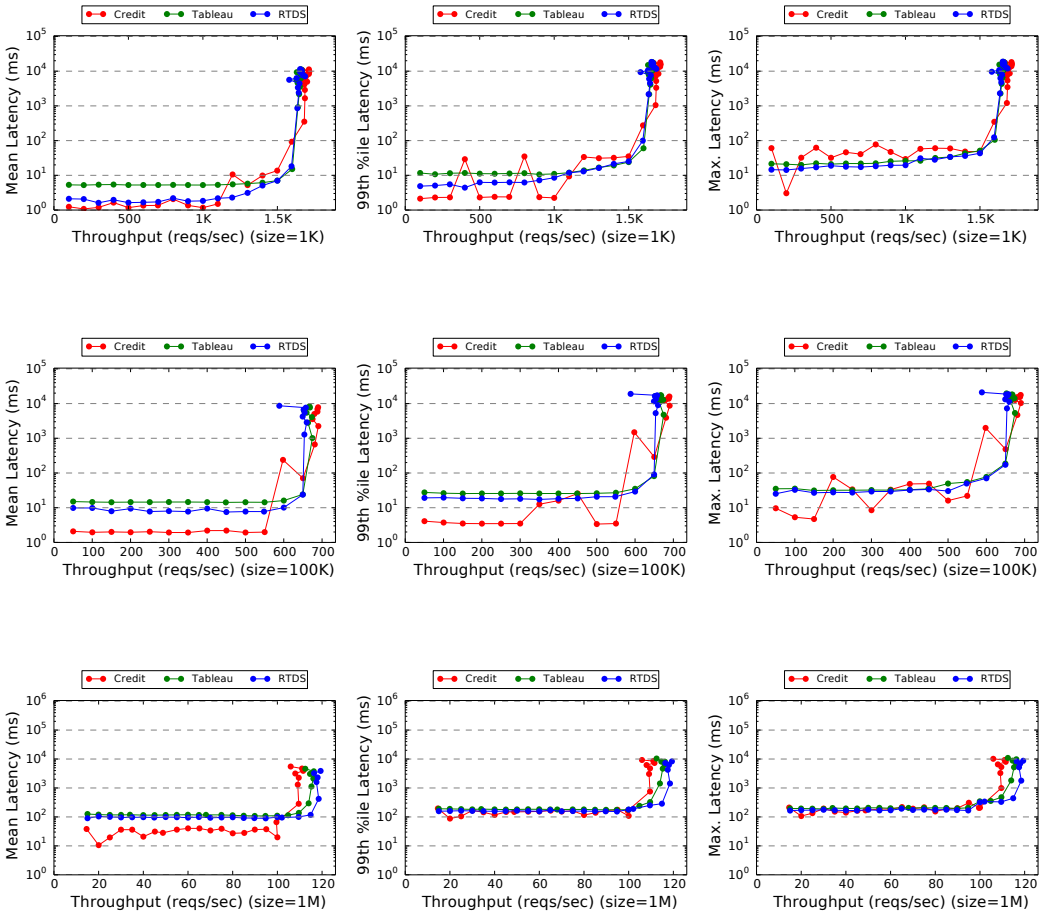


FIGURE B.16: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 15ms timeslice (or scheduling latency) and varying throughput.

B.4.2 Cache-Intensive Background Workload

B.4.2.1 Capped Scenario

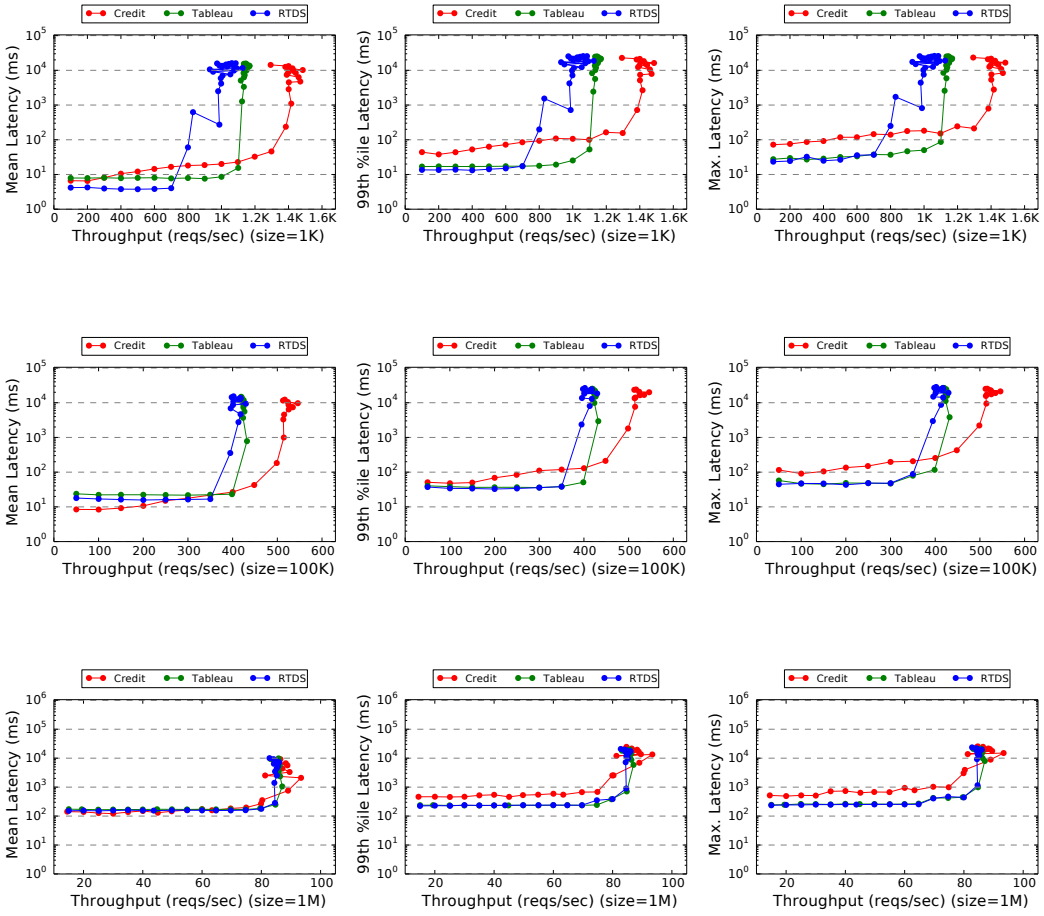


FIGURE B.17: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 15ms timeslice (or scheduling latency) and varying throughput.

B.4.2.2 Uncapped Scenario

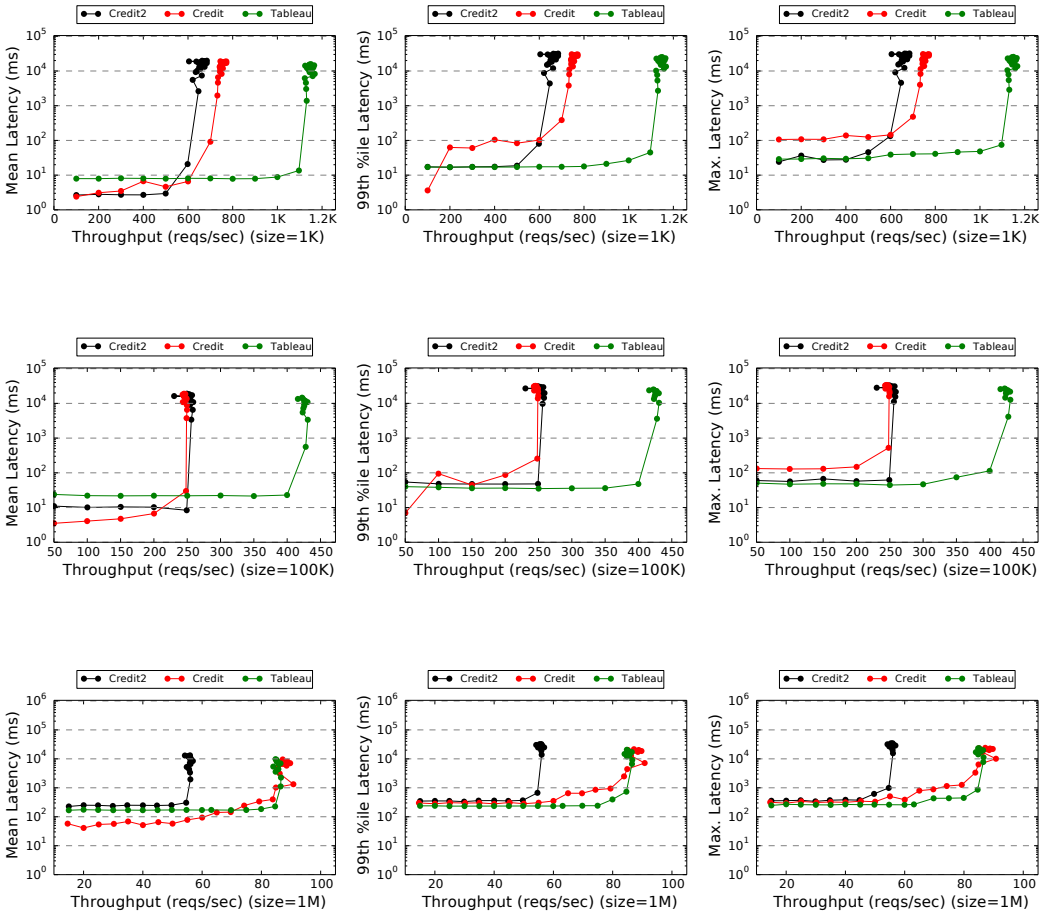


FIGURE B.18: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 15ms timeslice (or scheduling latency) and varying throughput.

B.4.3 I/O-Intensive Background Workload

B.4.3.1 Capped Scenario

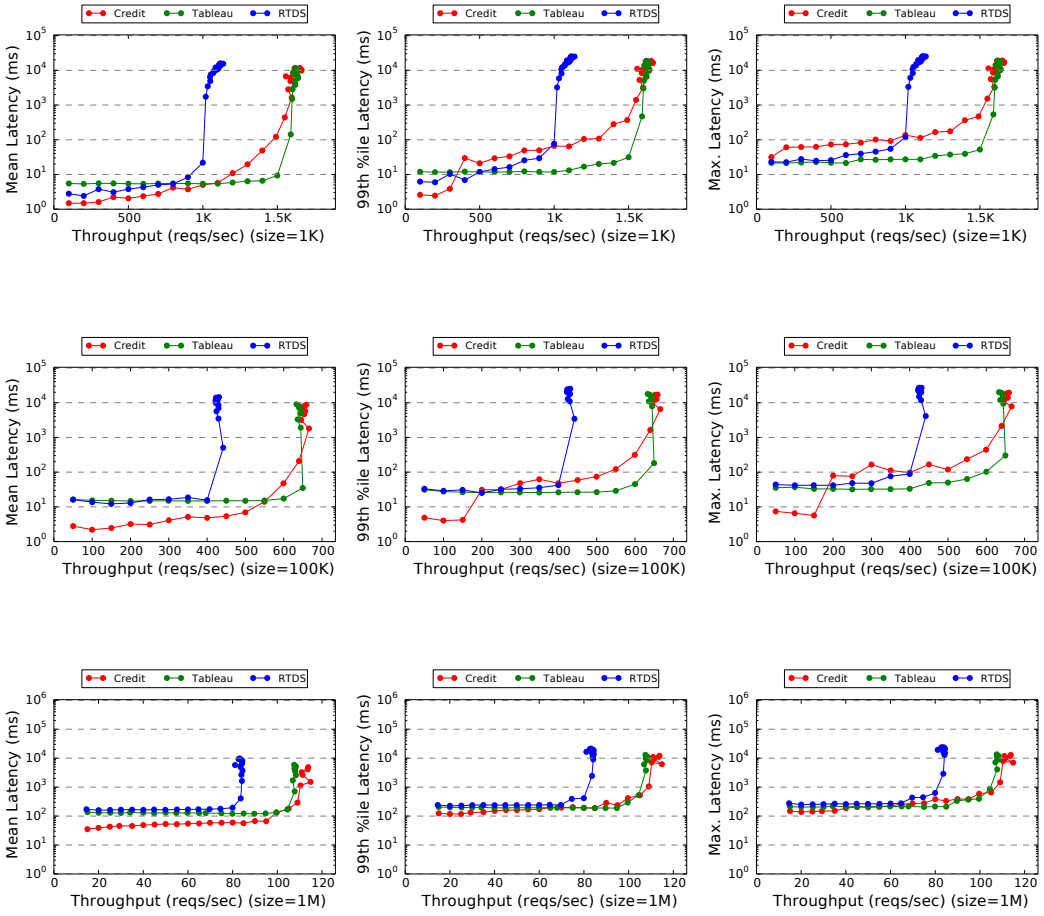


FIGURE B.19: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 15ms timeslice (or scheduling latency) and varying throughput.

B.4.3.2 Uncapped Scenario

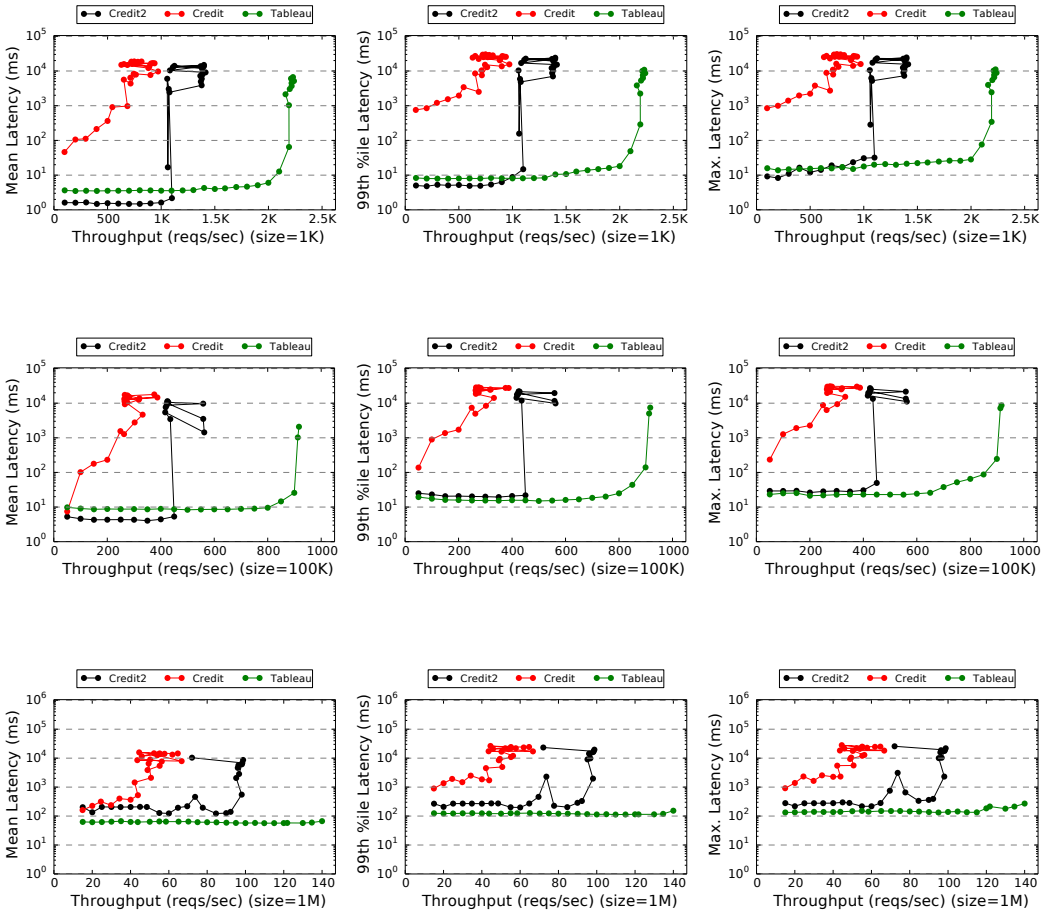


FIGURE B.20: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 15ms timeslice (or scheduling latency) and varying throughput.

B.5 20ms Scheduling Latency

B.5.1 Idle Background Workload

B.5.1.1 Capped Scenario

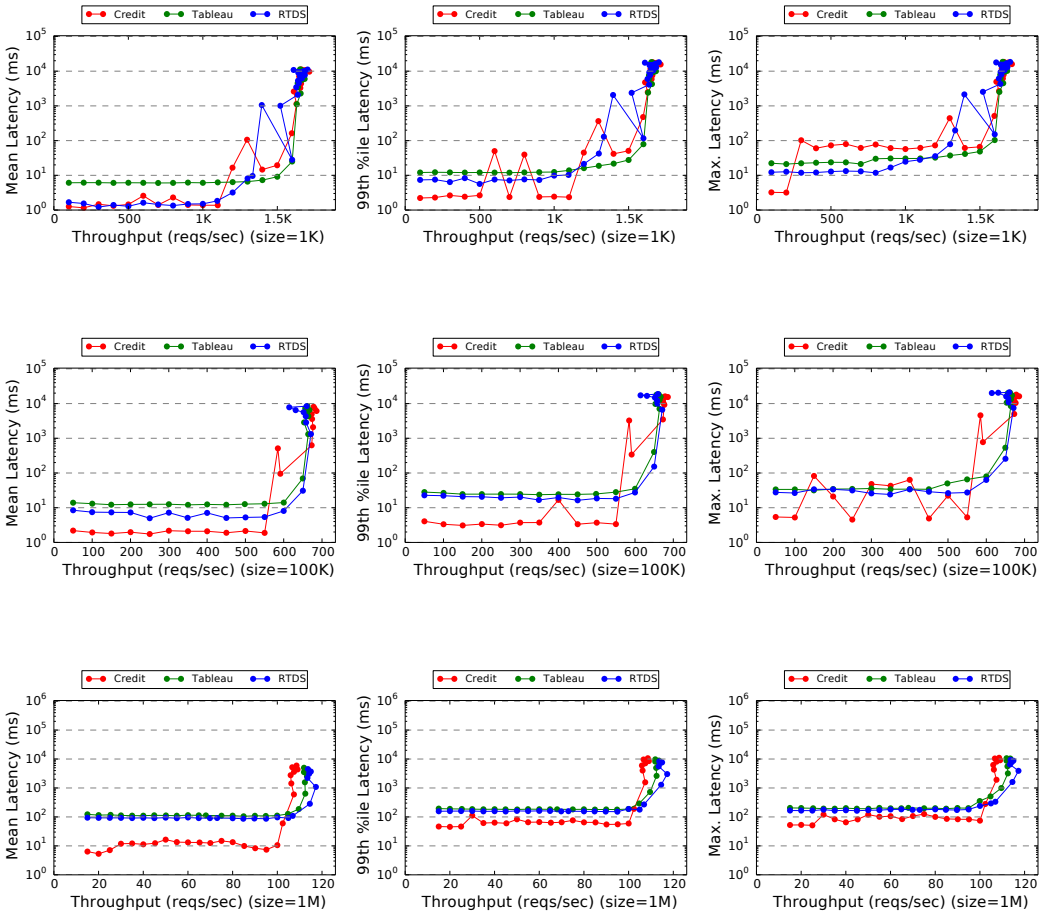


FIGURE B.21: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 20ms timeslice (or scheduling latency) and varying throughput.

B.5.2 Cache-Intensive Background Workload

B.5.2.1 Capped Scenario

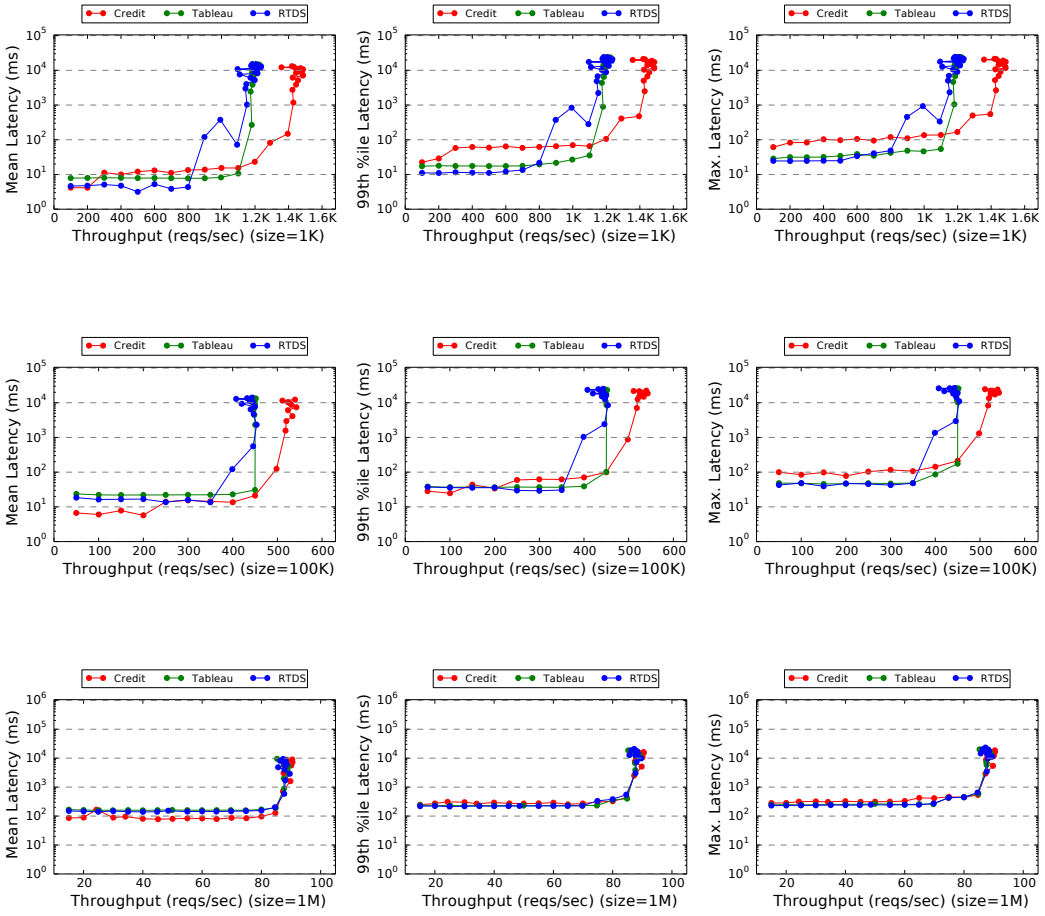


FIGURE B.22: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 20ms timeslice (or scheduling latency) and varying throughput.

B.5.2.2 Uncapped Scenario

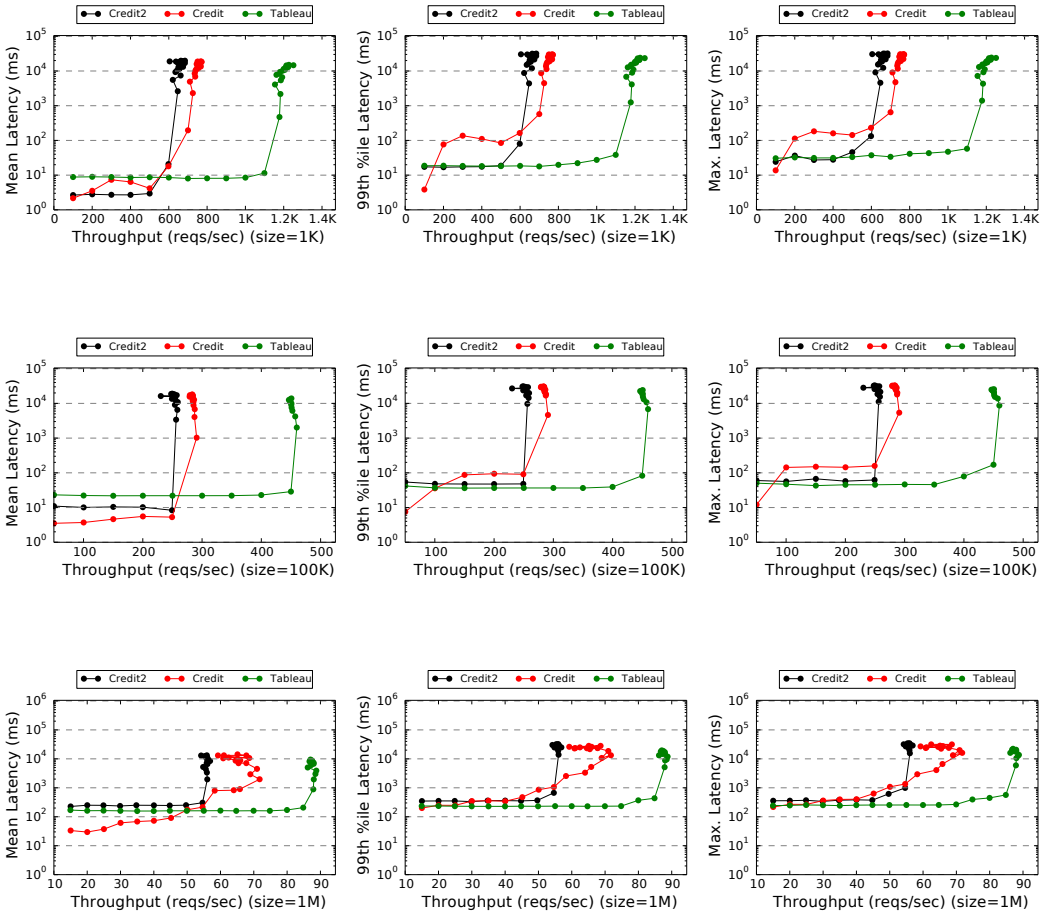


FIGURE B.23: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 20ms timeslice (or scheduling latency) and varying throughput.

B.5.3 I/O-Intensive Background Workload

B.5.3.1 Capped Scenario

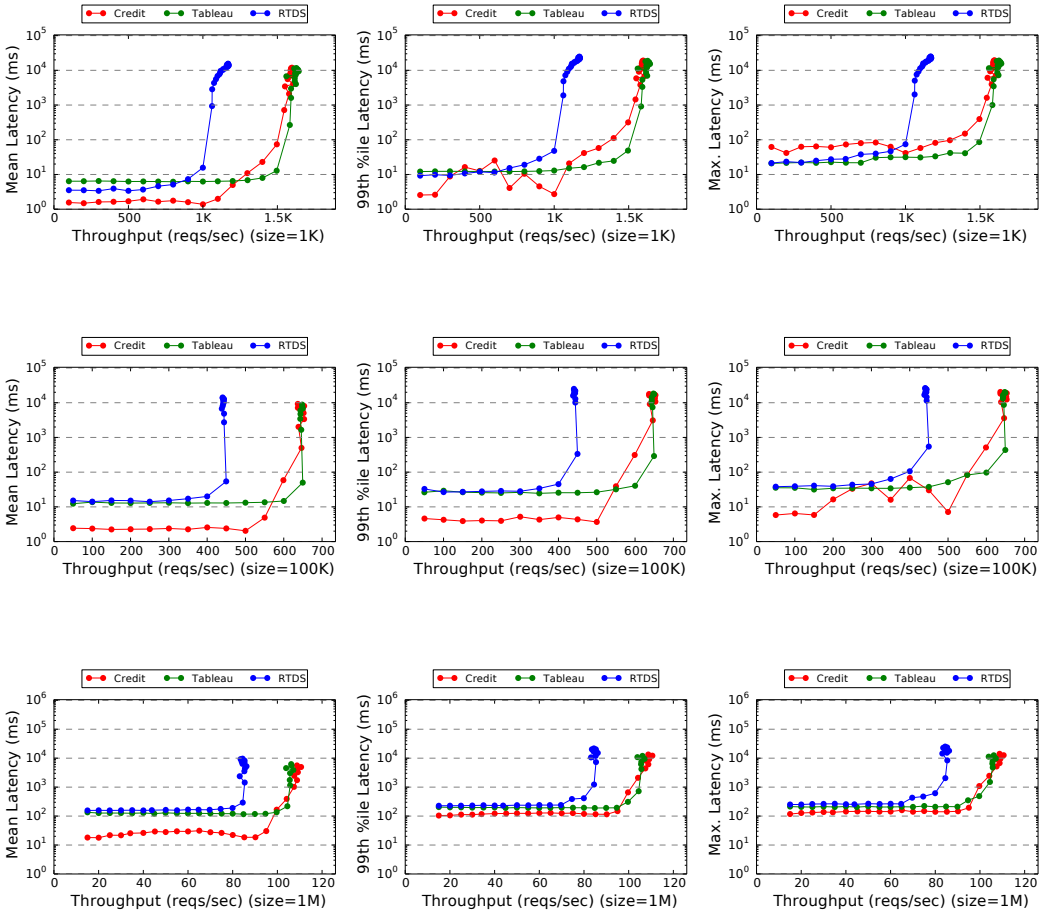


FIGURE B.24: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 20ms timeslice (or scheduling latency) and varying throughput.

B.5.3.2 Uncapped Scenario

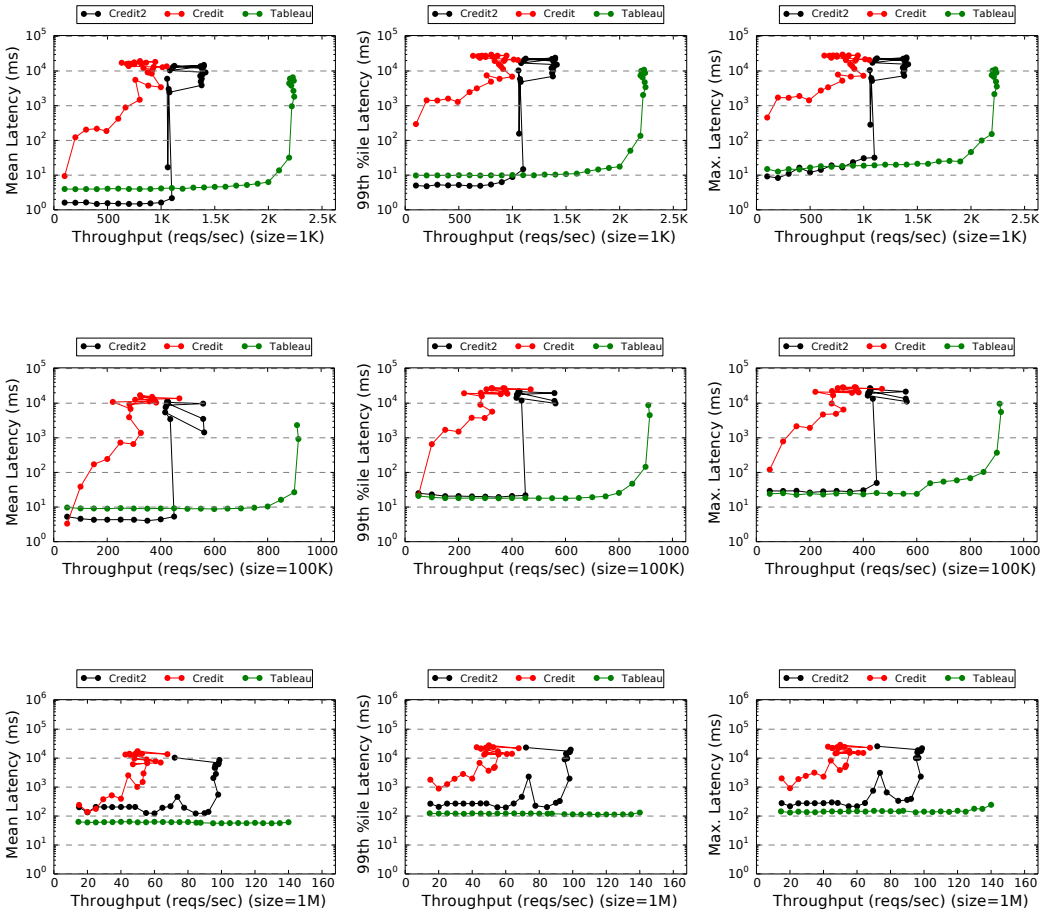


FIGURE B.25: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 20ms timeslice (or scheduling latency) and varying throughput.

B.6 25ms Scheduling Latency

B.6.1 Idle Background Workload

B.6.1.1 Capped Scenario

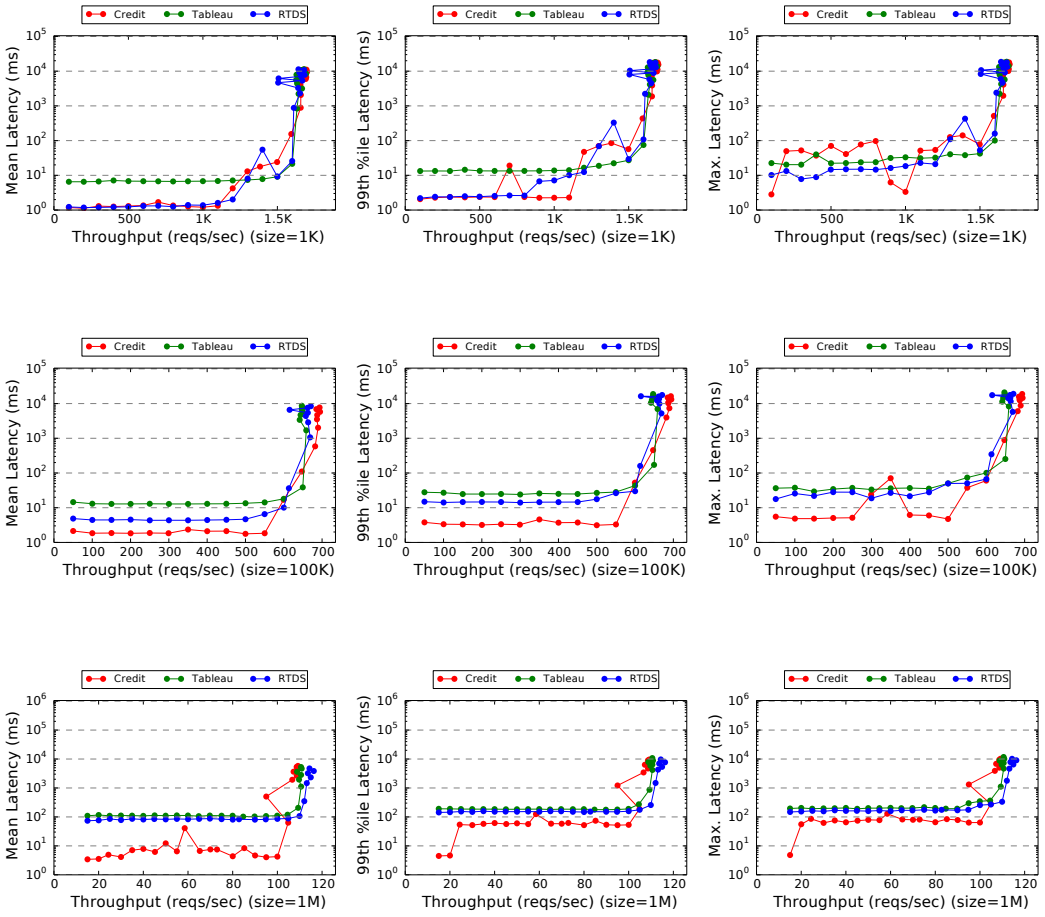


FIGURE B.26: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 25ms timeslice (or scheduling latency) and varying throughput.

B.6.2 Cache-Intensive Background Workload

B.6.2.1 Capped Scenario

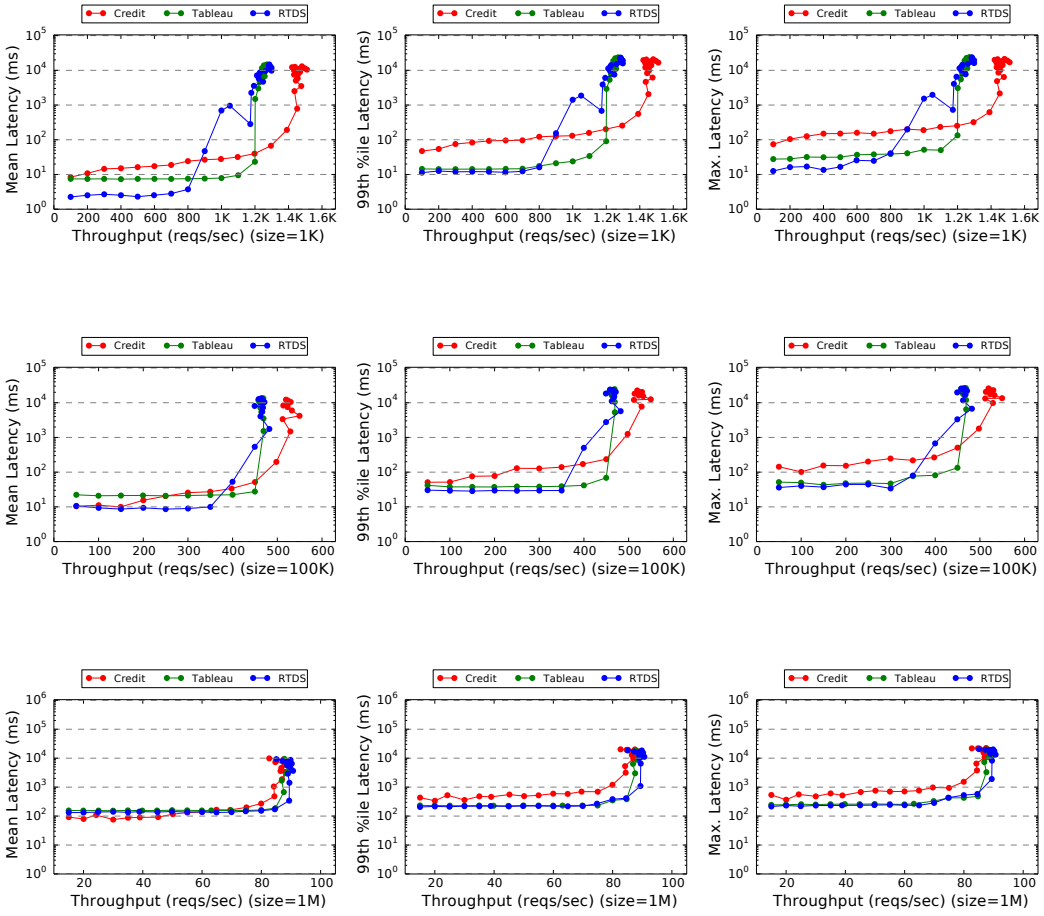


FIGURE B.27: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 25ms timeslice (or scheduling latency) and varying throughput.

B.6.2.2 Uncapped Scenario

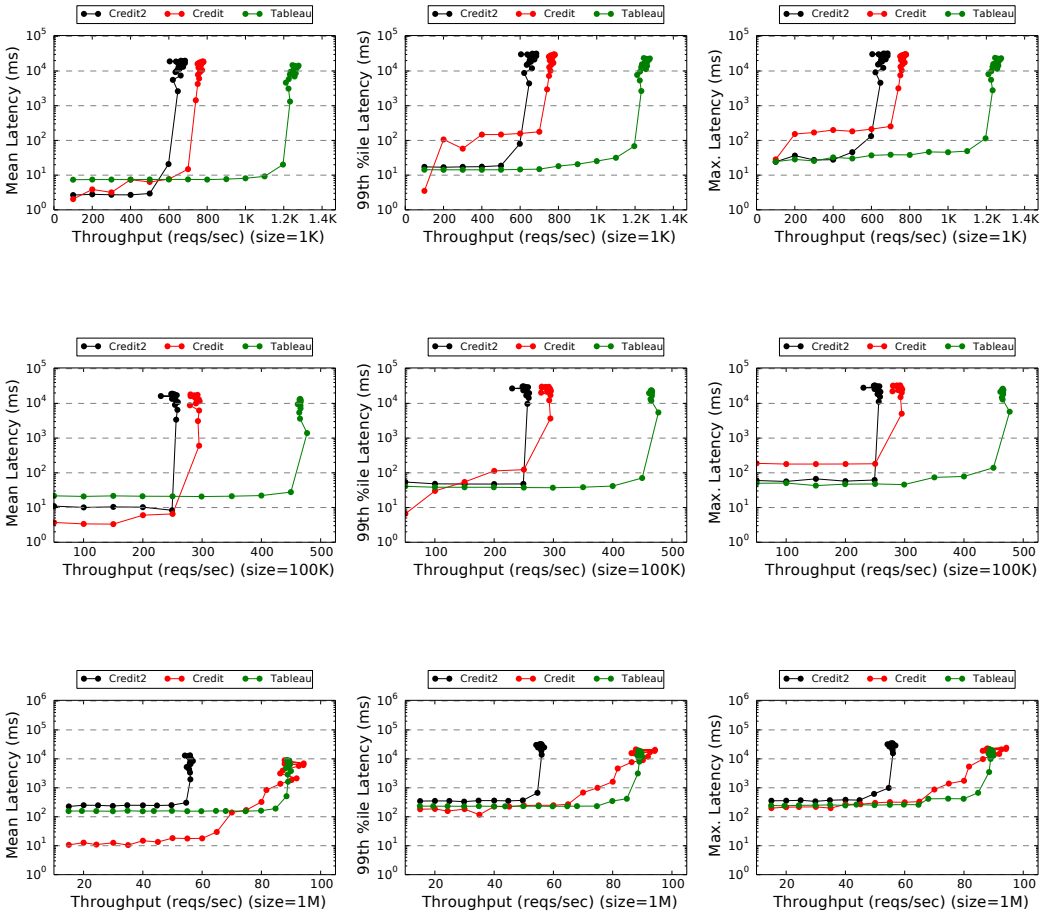


FIGURE B.28: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 25ms timeslice (or scheduling latency) and varying throughput.

B.6.3 I/O-Intensive Background Workload

B.6.3.1 Capped Scenario

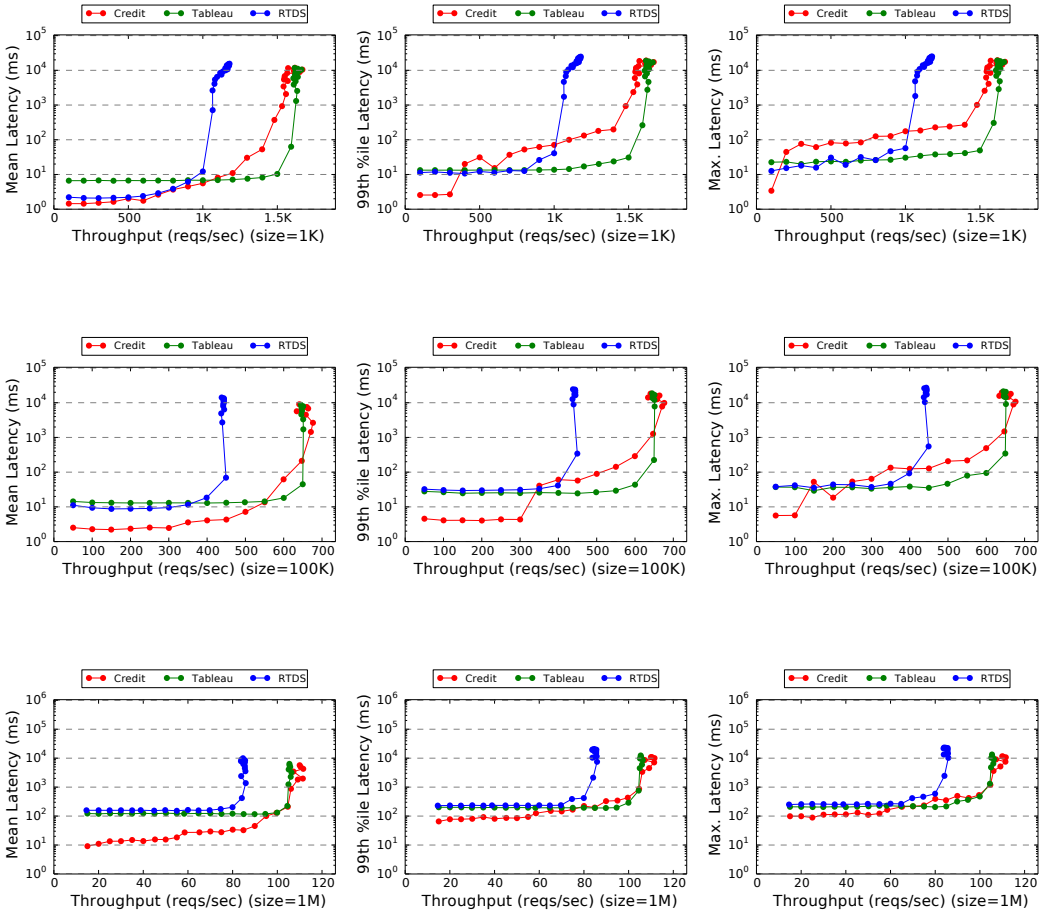


FIGURE B.29: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 25ms timeslice (or scheduling latency) and varying throughput.

B.6.3.2 Uncapped Scenario

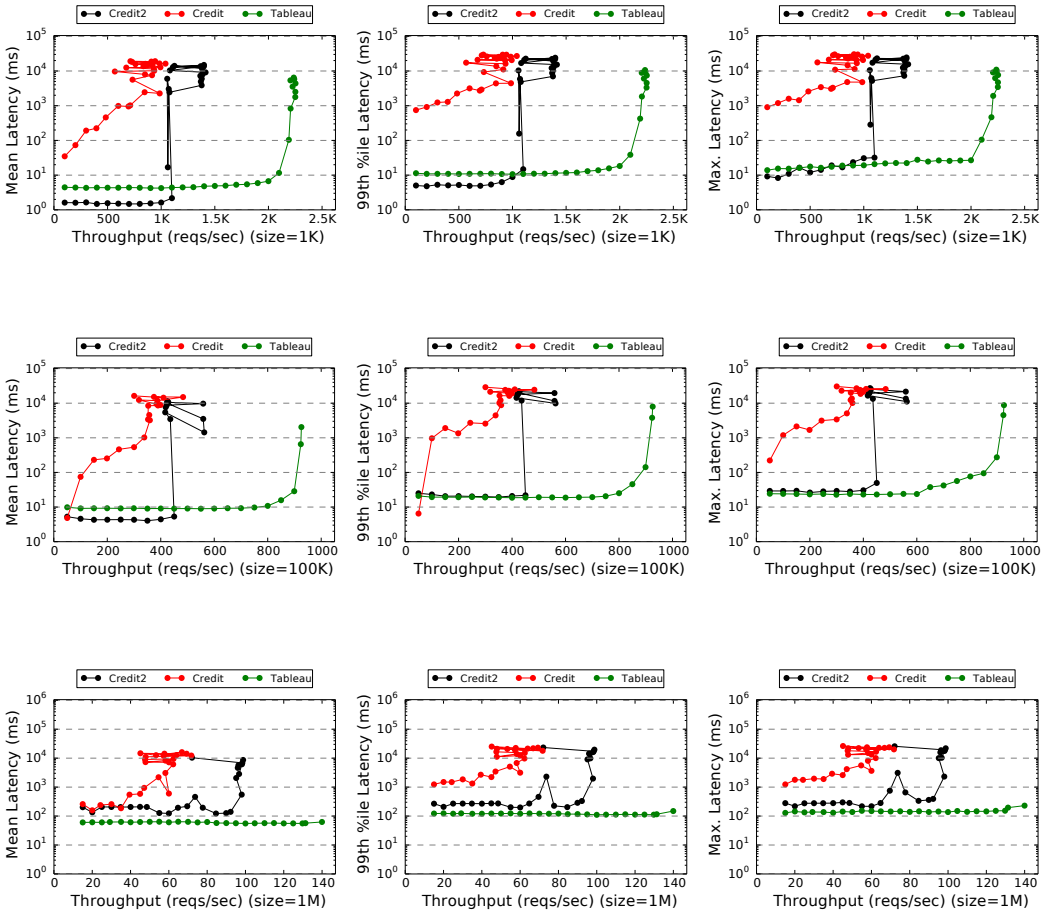


FIGURE B.30: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 25ms timeslice (or scheduling latency) and varying throughput.

B.7 30ms Scheduling Latency

B.7.1 Idle Background Workload

B.7.1.1 Capped Scenario

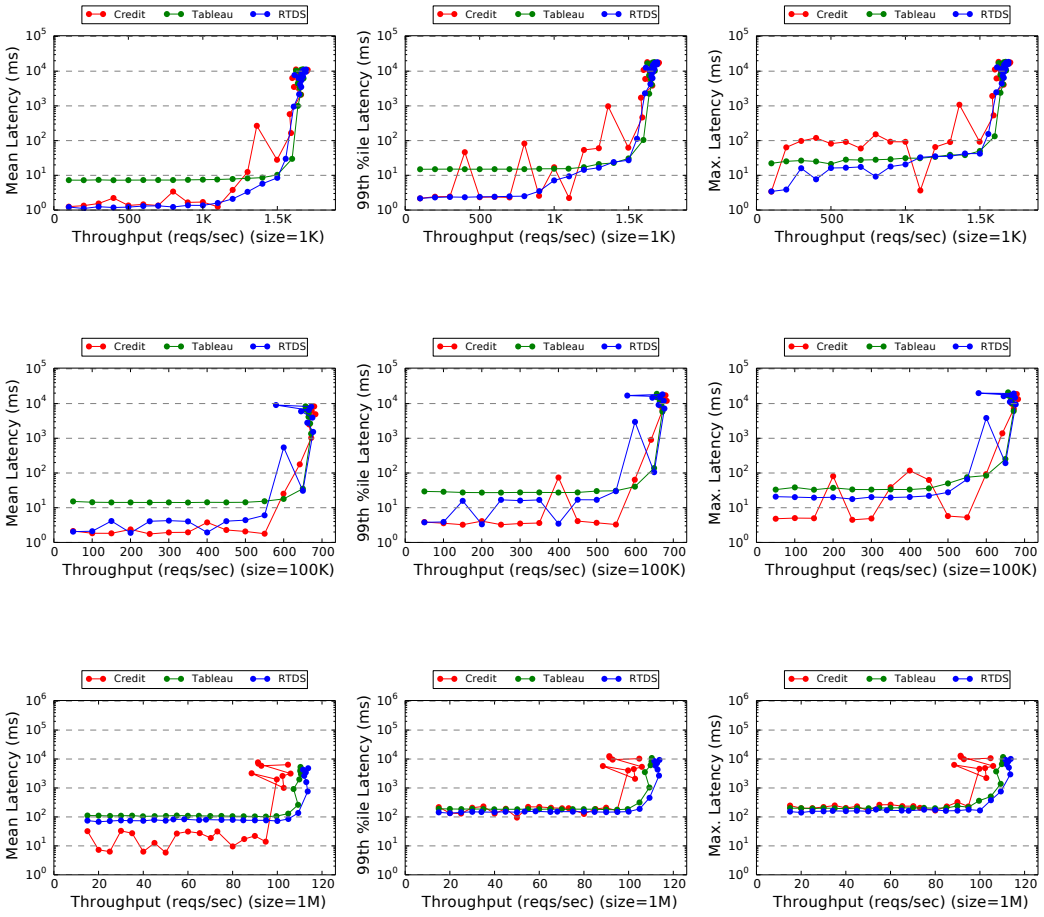


FIGURE B.31: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 30ms timeslice (or scheduling latency) and varying throughput.

B.7.2 Cache-Intensive Background Workload

B.7.2.1 Capped Scenario

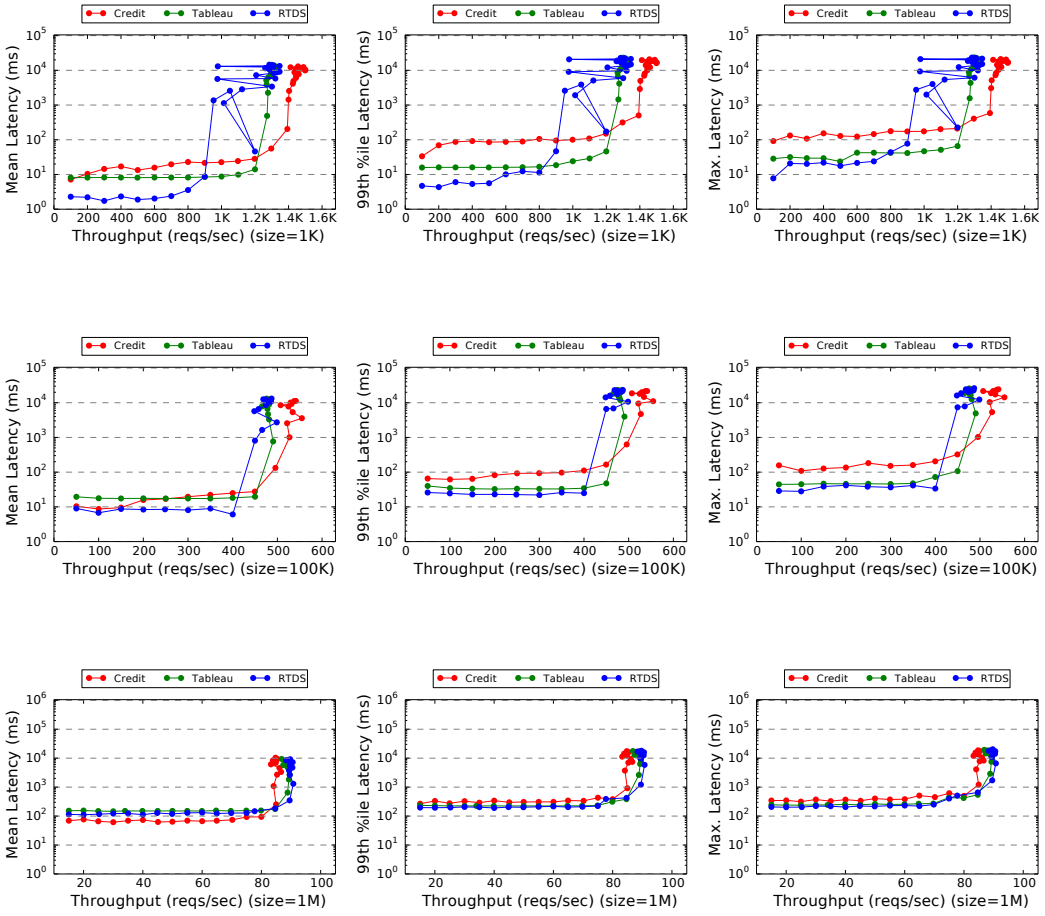


FIGURE B.32: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 30ms timeslice (or scheduling latency) and varying throughput.

B.7.2.2 Uncapped Scenario

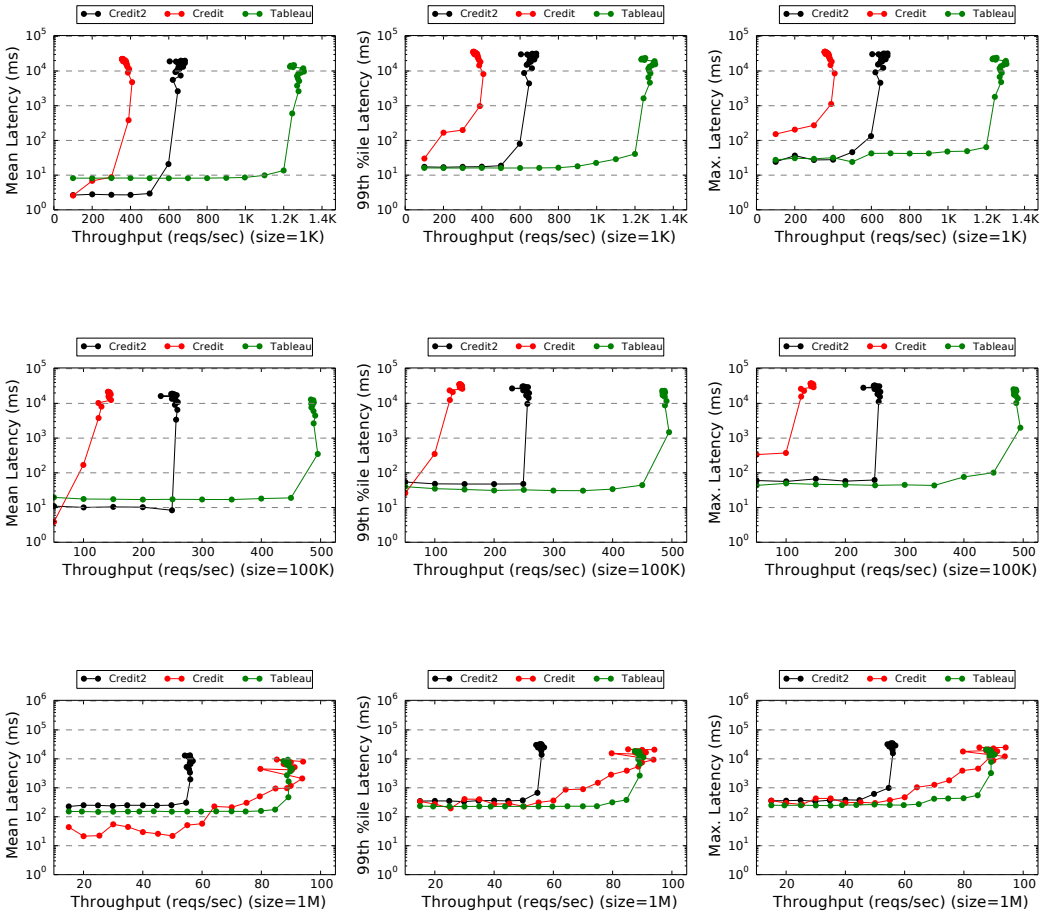


FIGURE B.33: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 30ms timeslice (or scheduling latency) and varying throughput.

B.7.3 I/O-Intensive Background Workload

B.7.3.1 Capped Scenario

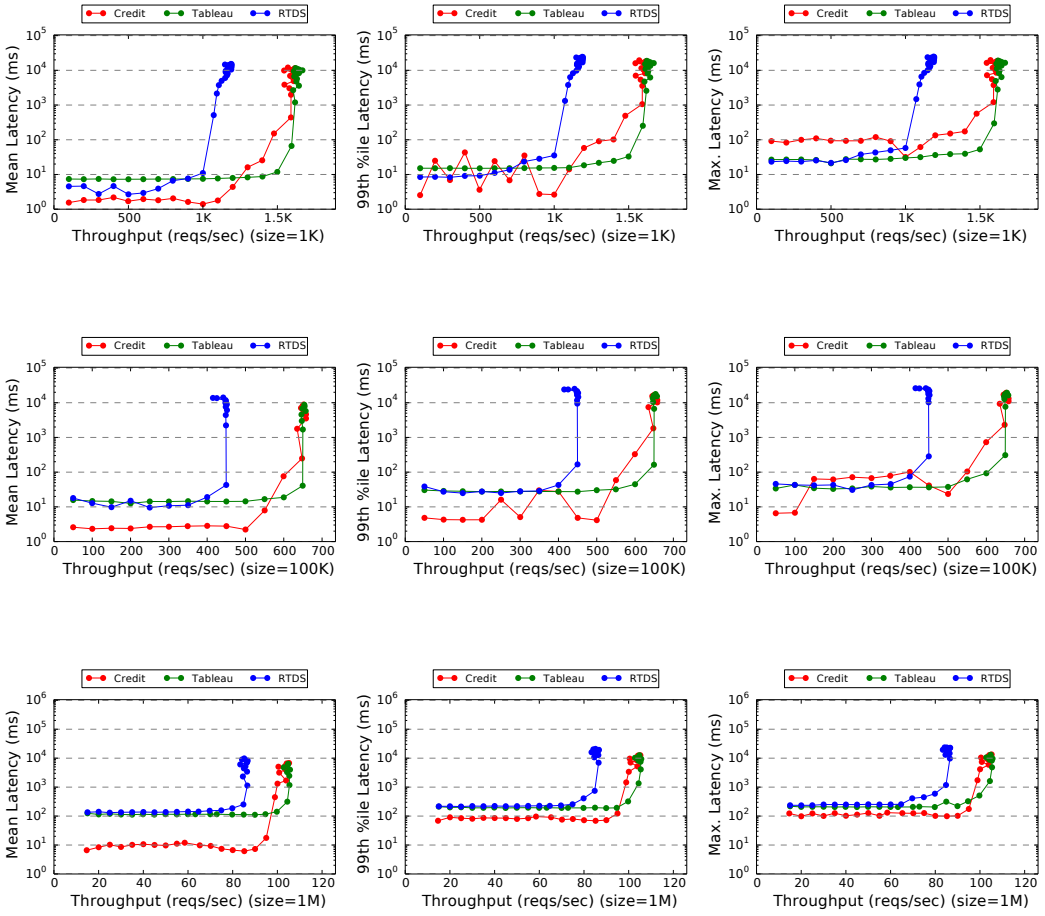


FIGURE B.34: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 30ms timeslice (or scheduling latency) and varying throughput.

B.7.3.2 Uncapped Scenario

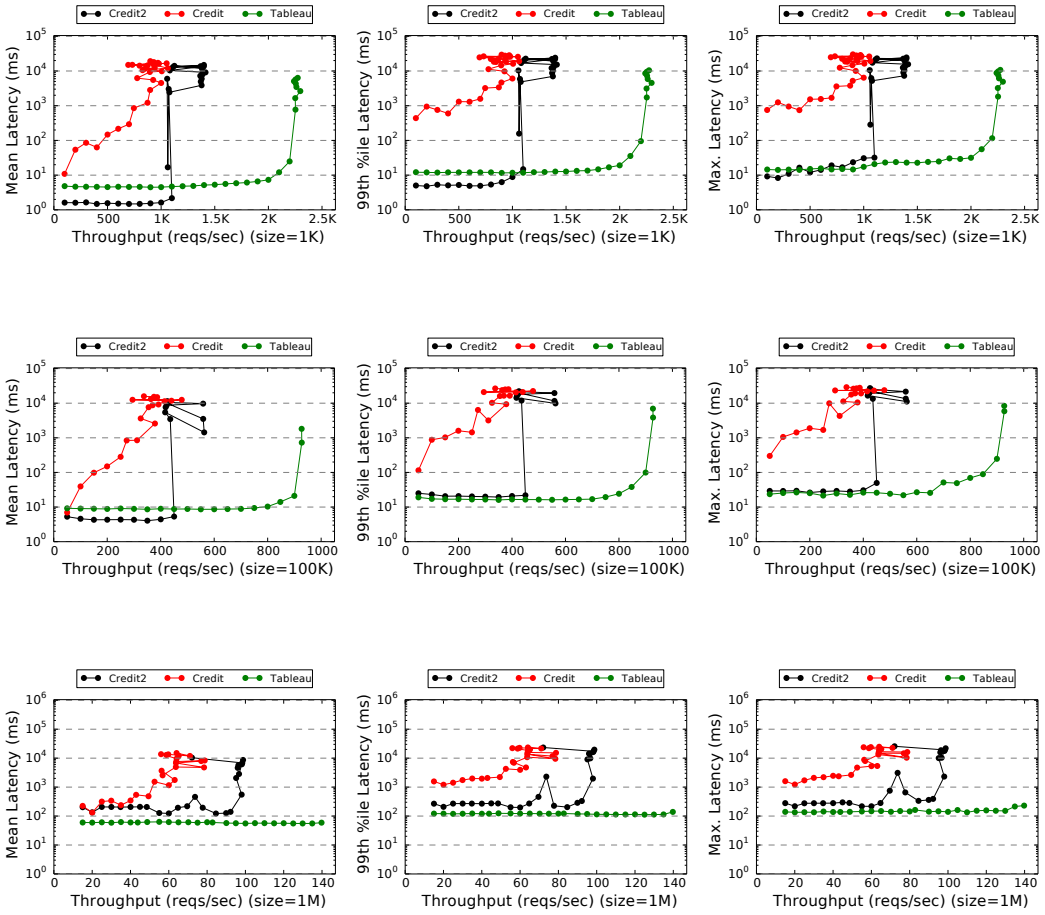


FIGURE B.35: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 30ms timeslice (or scheduling latency) and varying throughput.

B.8 35ms Scheduling Latency

B.8.1 Idle Background Workload

B.8.1.1 Capped Scenario

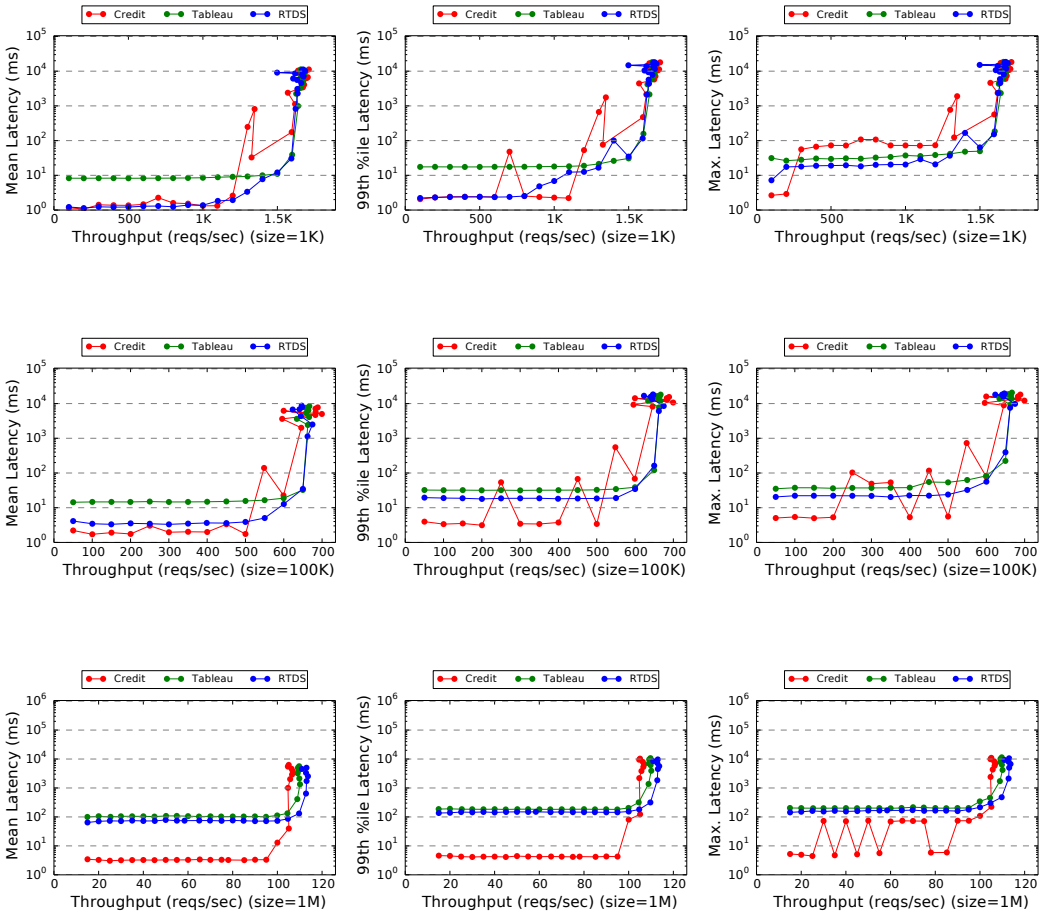


FIGURE B.36: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with a 35ms timeslice (or scheduling latency) and varying throughput.

B.8.2 Cache-Intensive Background Workload

B.8.2.1 Capped Scenario

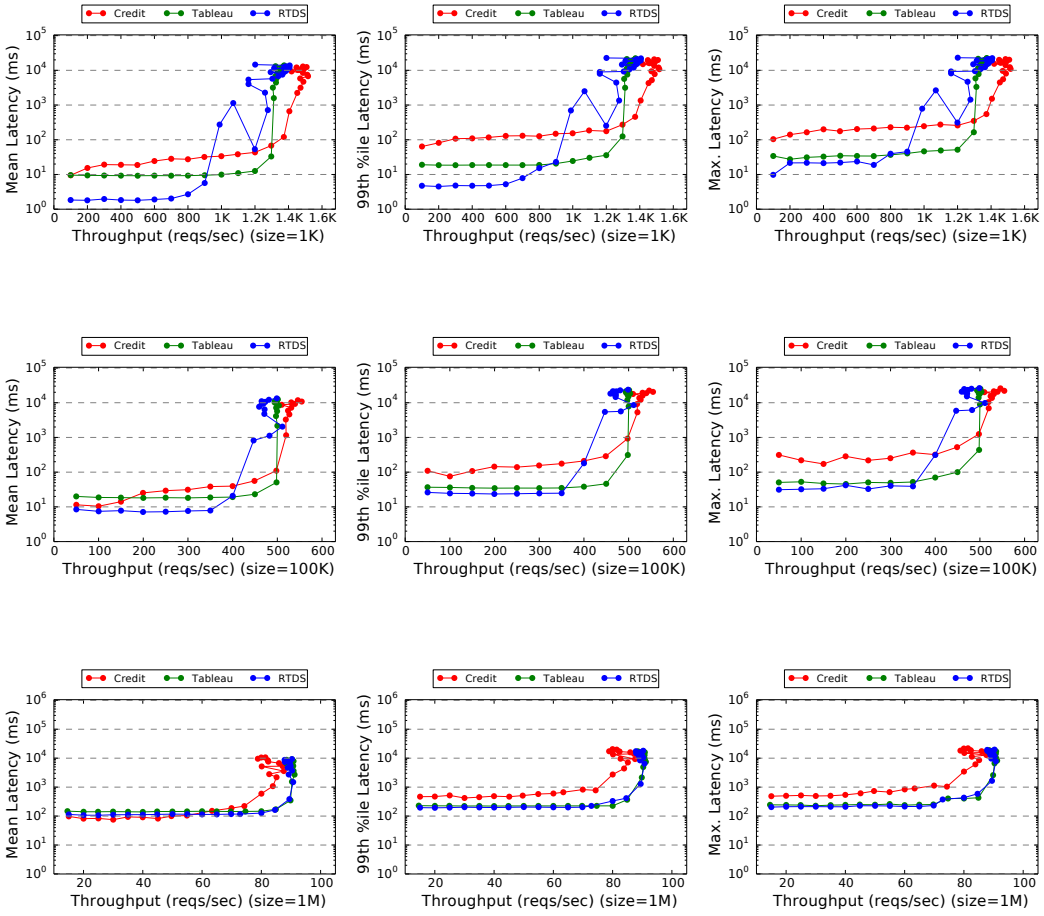


FIGURE B.37: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 35ms timeslice (or scheduling latency) and varying throughput.

B.8.2.2 Uncapped Scenario

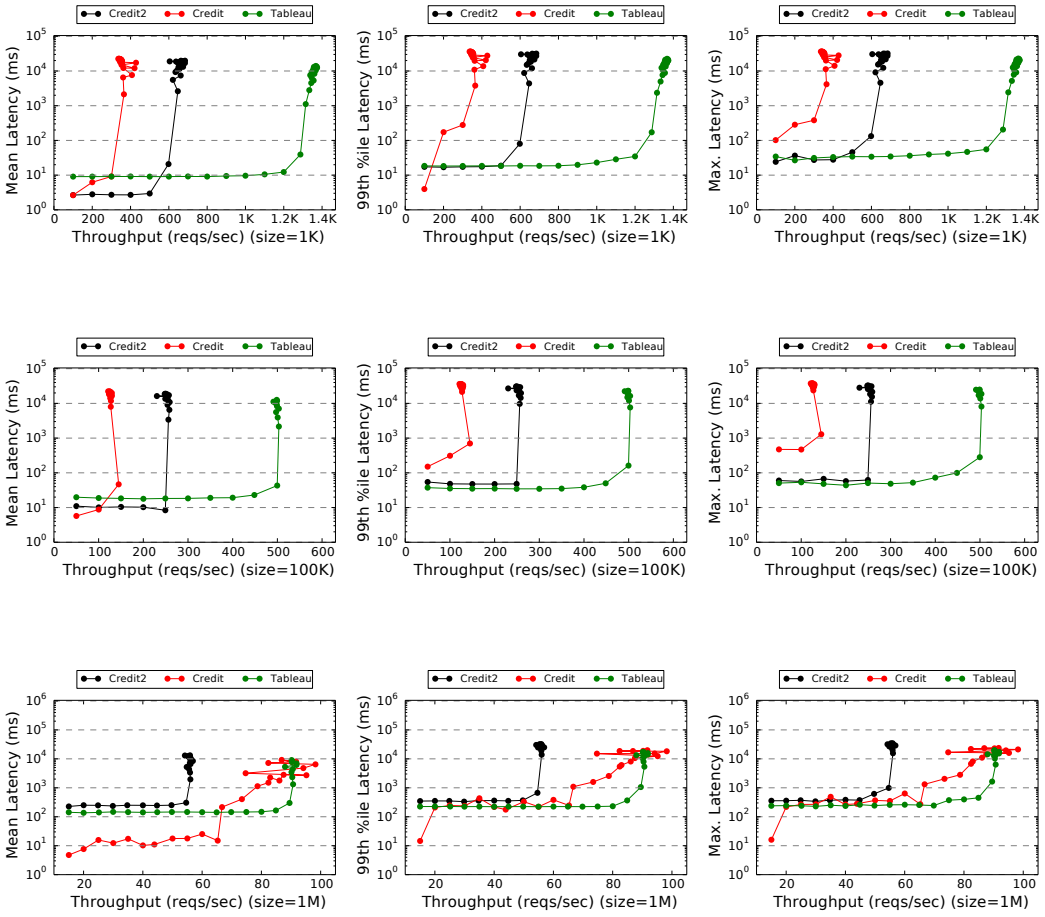


FIGURE B.38: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with a 35ms timeslice (or scheduling latency) and varying throughput.

B.8.3 I/O-Intensive Background Workload

B.8.3.1 Capped Scenario

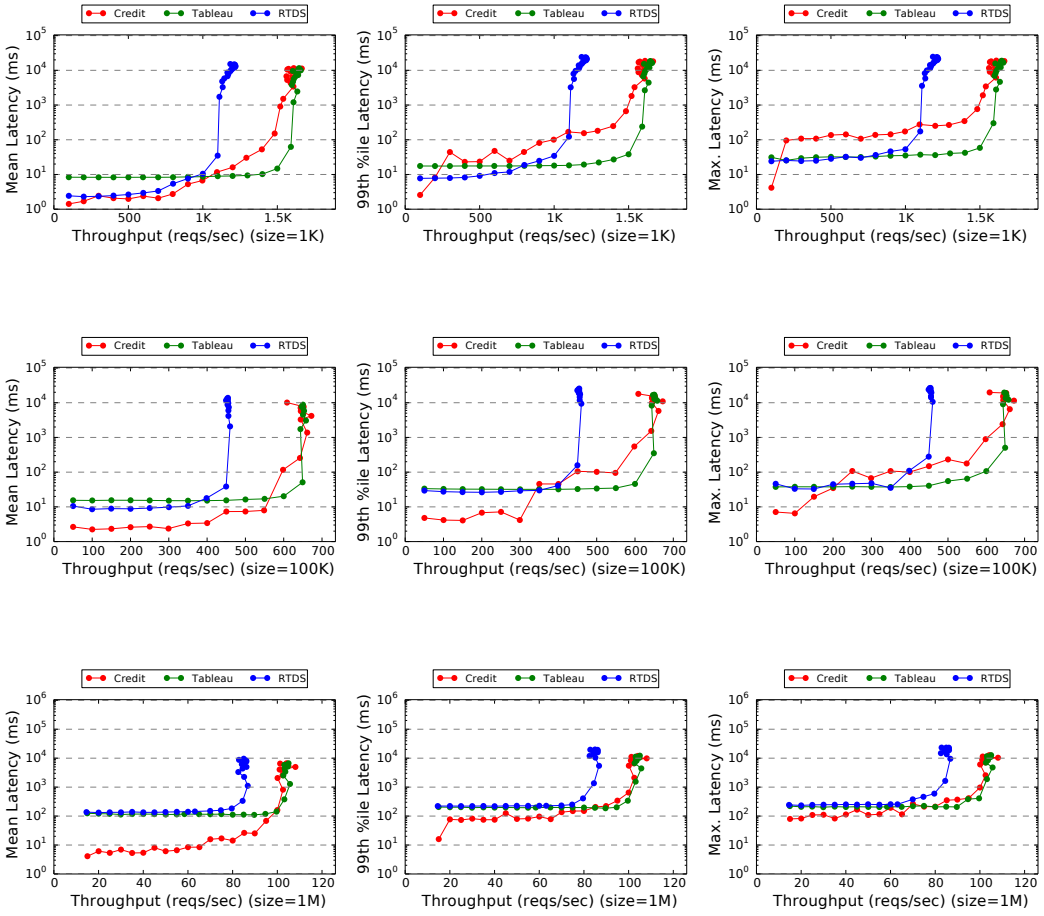


FIGURE B.39: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, RTDS, and Tableau under a capped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 35ms timeslice (or scheduling latency) and varying throughput.

B.8.3.2 Uncapped Scenario

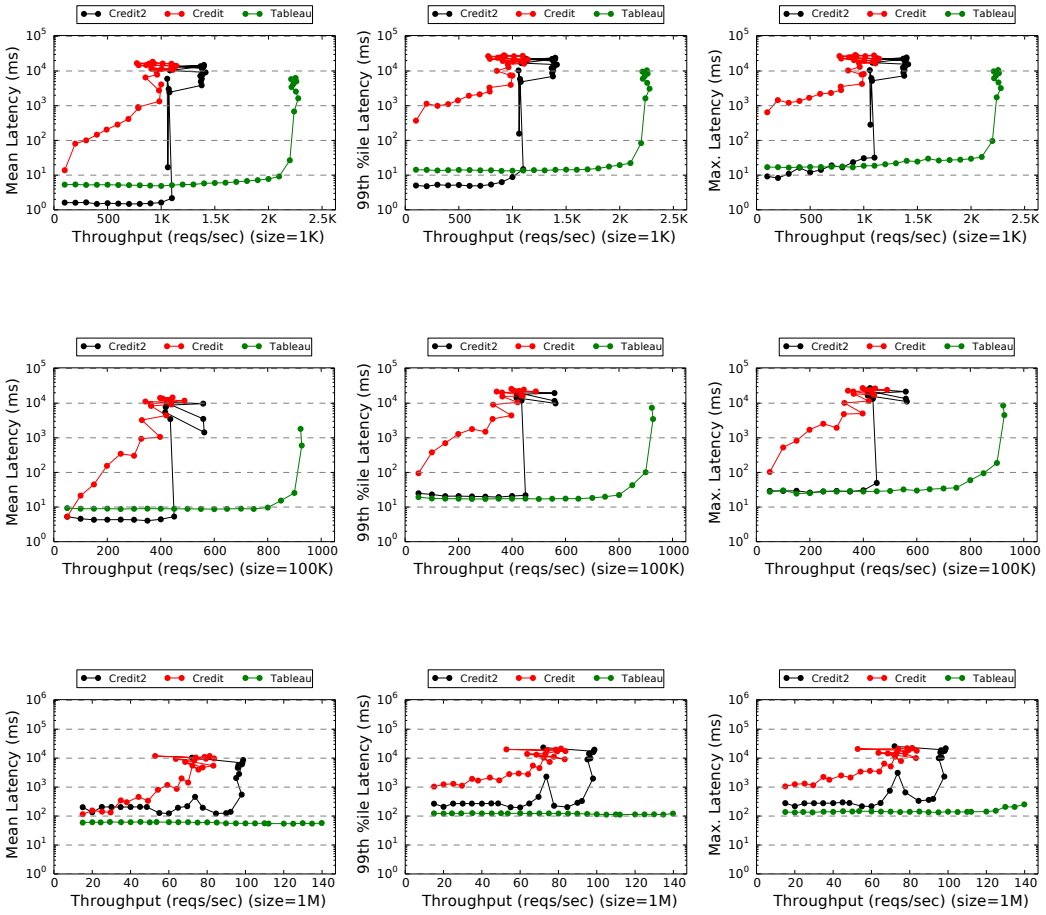


FIGURE B.40: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, and Tableau under an uncapped scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with a 35ms timeslice (or scheduling latency) and varying throughput.

Appendix C

Extended Evaluation: Dedicated-Core Performance Comparison

This section compares the performance of the four evaluated Xen schedulers (Credit, Credit2, RTDS, and Tableau) in a full-core scenario. That is, each VM is assigned an entire core with only twelve VMs being present in the system (one for each available core on our 16-core machine with four cores dedicated to Dom0).

When dedicating an entire core to a VM, configuring scheduling latency has no effect since the scheduler does not need to multiplex VMs on each core. Therefore each scheduler was configured using default parameters (or recommended parameters, when available). For Tableau and RTDS, a 5 ms scheduling latency was chosen, while a 5 ms global timeslice was configured under Credit. Credit2 did not support any configuration of the scheduling latency at the time of writing this thesis. Consequently, four sections show the results comparing the performance of these schedulers under four background workloads (idle, CPU-intensive, cache-intensive, and I/O-intensive), respectively.

C.1 Idle Background Workload

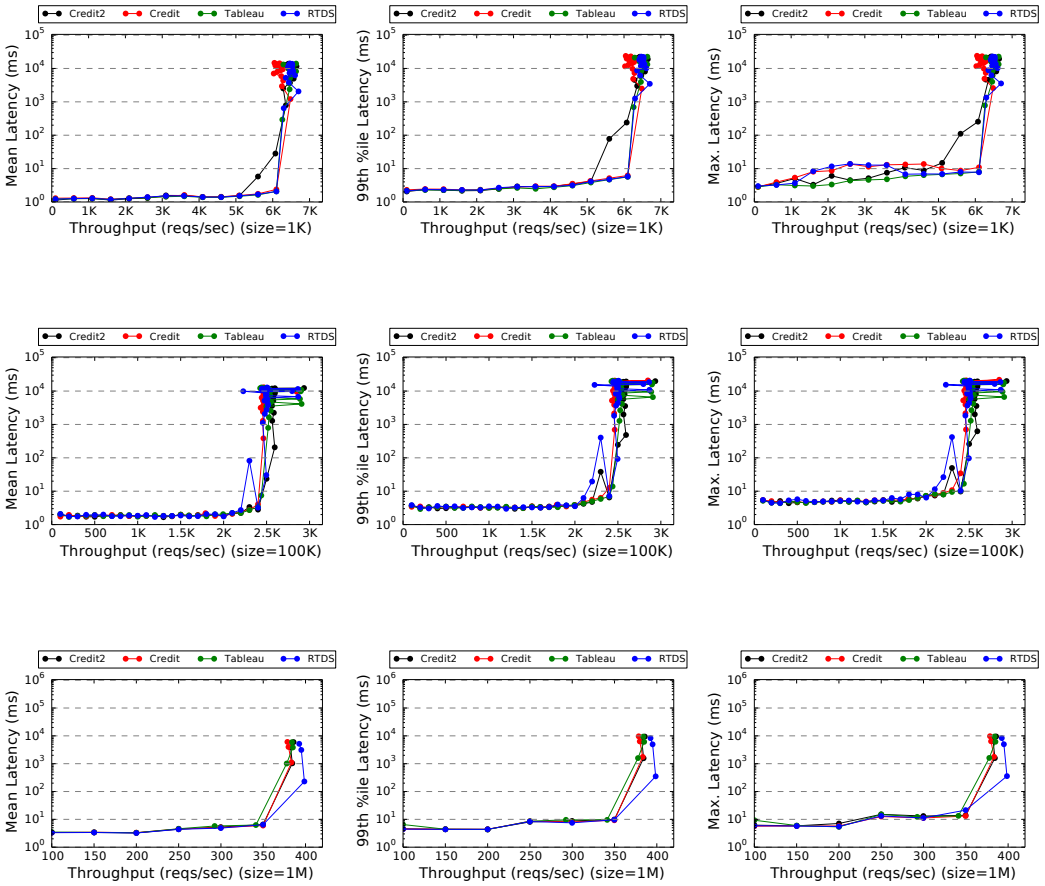


FIGURE C.1: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, RTDS, and Tableau under a dedicated-core scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an idle background workload and with varying throughput.

C.2 CPU-Intensive Background Workload

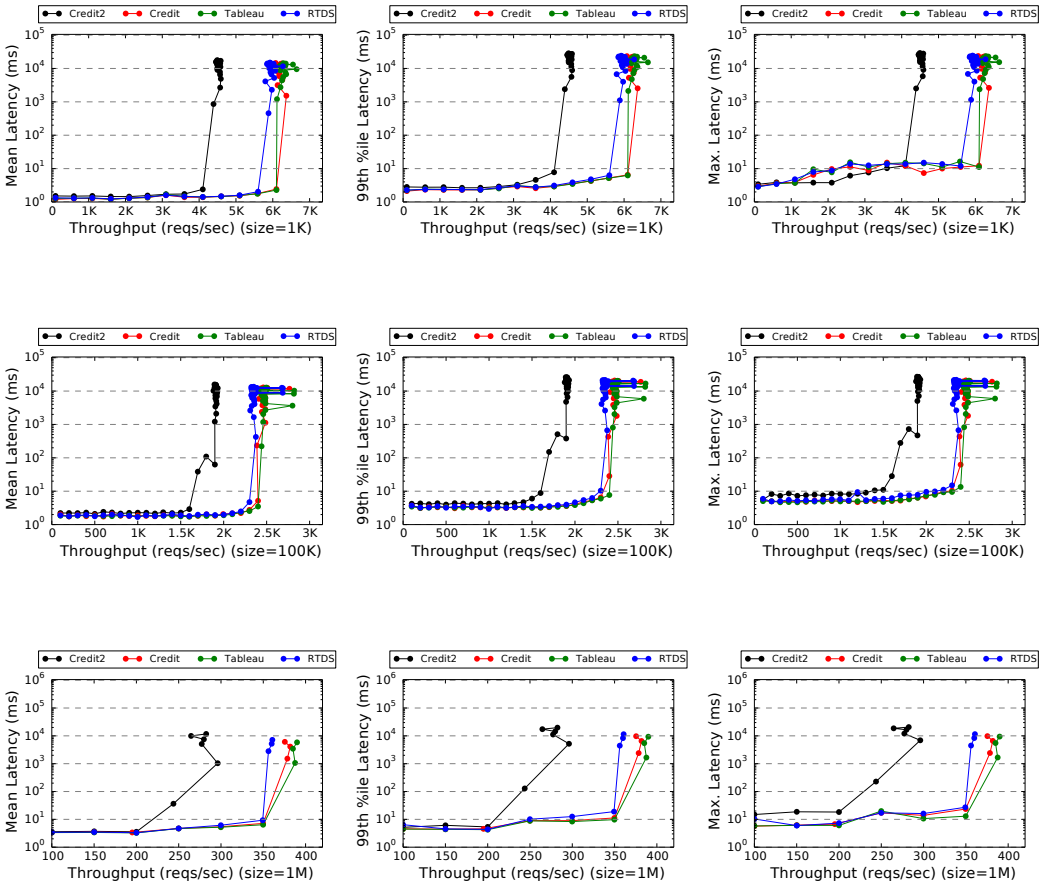


FIGURE C.2: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, RTDS, and Tableau under a dedicated-core scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a CPU-intensive background workload and with varying throughput.

C.3 Cache-Intensive Background Workload

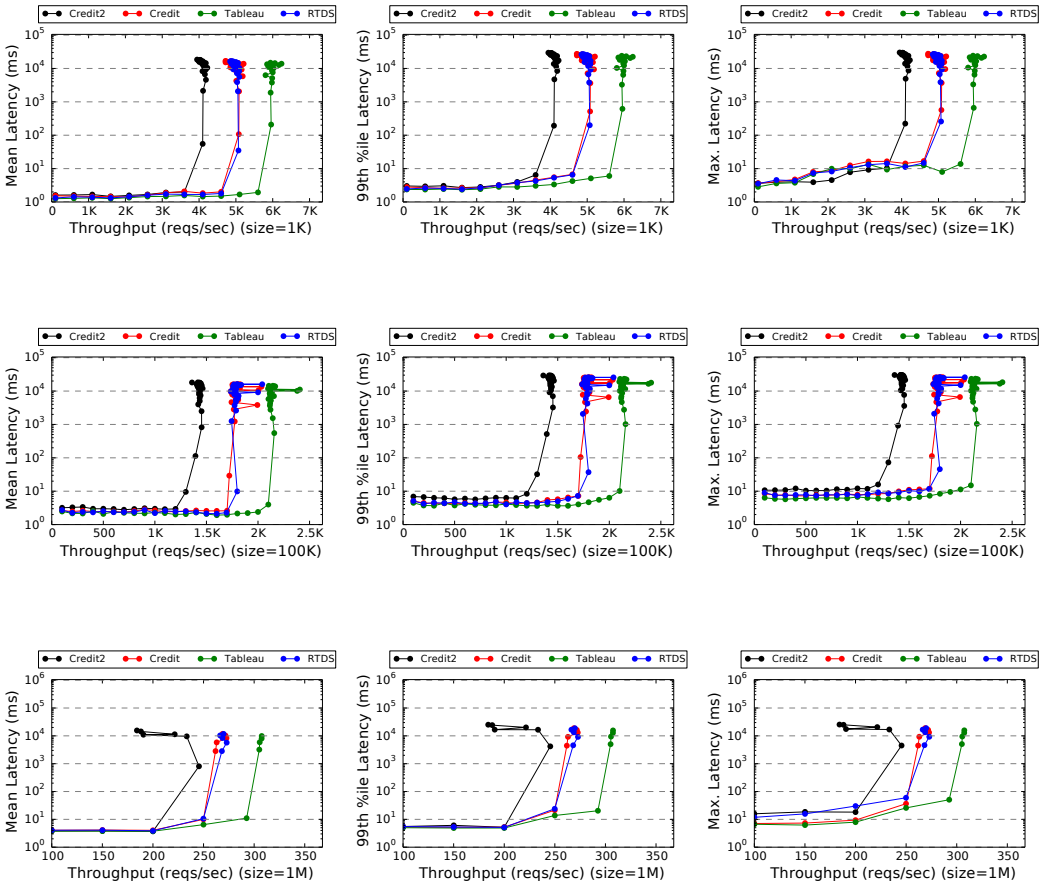


FIGURE C.3: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, RTDS, and Tableau under a dedicated-core scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with a cache-intensive background workload and with varying throughput.

C.4 I/O-Intensive Background Workload

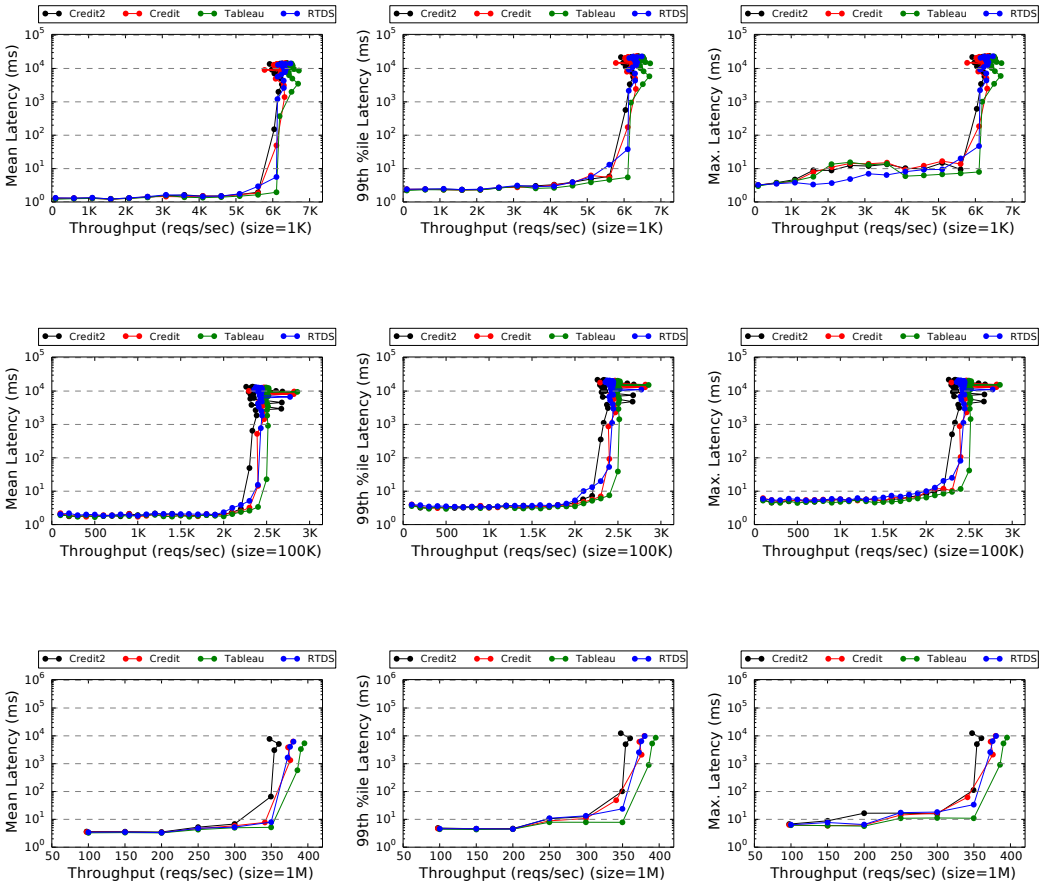


FIGURE C.4: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for Credit, Credit2, RTDS, and Tableau under a dedicated-core scenario, requesting 1 KiB, 100 KiB, and 1 MiB files, with an I/O-intensive background workload and with varying throughput.

Appendix D

Tableau Table Generation Script

Listing D.1 through Listing D.10 show various Python functions that are used to generate the binary structure of the scheduling table that the Tableau scheduler uses. The entry point for the code below is the `pack_global()` function shown in Listing D.1.

```
1  # Pack the entire scheduling table into a binary blob.
2  def pack_global(vcpu_list, l1_schedule, l1_slices, l1_slice_lens):
3      num_cpus = len(l1_schedule['tables'])
4
5      # 1. Pack a VCPU info list containing a list of all vCPUs
6      #    in the system.
7      vinfo_packed = pack_vcpu_info(vcpu_list)
8
9      # 2. Pack a per-CPU structure for each CPU in the system,
10     #    containing the list of local vCPUs, the schedule,
11     #    and the slice table.
12     percpu_packed = []
13     for i in xrange(num_cpus):
14         percpu_struct = pack_percpu(i, vcpu_list,
15                                     l1_schedule['tables'],
16                                     l1_slices, l1_slice_lens[i],
17                                     l1_schedule['length'])
18         percpu_packed.append(percpu_struct)
19
20     # 3. Pack a global header containing metadata about the
```

```
21      #     rest of the table. (1) The number of vCPUs in the
22      #     table, the numbre of CPUs for which a schedule is
23      #     provided.
24      #
25      #     The header is immediately followed by the packed
26      #     list of vCPUs in the system, followed by a packed
27      #     structure for each CPU in the system.
28      header = pack_global_header(len(vcpu_list),
29                                  num_cpus,
30                                  vinfo_packed,
31                                  percpu_packed)
32
33      # The Pack() object generates a flattened binary of all
34      # the packed data appended to it. The flattening is done
35      # in the order of insertion of data.
36      ret = Pack()
37      ret.push_packed('global_header', header)
38      ret.push_packed('vinfo', vinfo_packed)
39      for i in xrange(num_cpus):
40          ret.push_packed('percpu[' + str(i) + ']', percpu_packed[i])
41
42      return ret
```

LISTING D.1: Python function for packing a Tableau scheduler table.

```
1   # Packs the global header containing metadata about the table.
2   def pack_global_header(num_vcpus, num_cpus, vinfo, percpu_packed):
3
4       ret = Pack()
5
6       # Push the number of CPUs
7       ret.push_u64('num_vcpus', num_vcpus)
8       # The global vCPU list is at the end of this structure.
9       # Since this structure is page aligned to 4KB, the vCPU
10      # list can be found at byte 4096.
11      ret.push_u64('vcpu_list_off', 4096)
12
13      # Push the number of physical CPUs in the system. This
```

```
14      # corresponds to the number of per-CPU structure appended
15      # after the vCPU list.
16      ret.push_u64('num_cpus', num_cpus)
17
18      # For each per-CPU structure, append the offset within
19      # the table where it can be found.
20      off = 4096 + vinfo.length()
21      for i in xrange(len(percpu_packed)):
22          ret.push_u64('percpu_off[' + str(i) + ']', off)
23          off += percpu_packed[i].length()
24
25      # Pad with zeros so a total of 64 entries are filled up.
26      for i in xrange(len(percpu_packed), 64):
27          ret.push_u64('percpu_off[' + str(i) + ']', 0)
28
29      # Align this structure to 4KB by padding with zeroes.
30      ret.align_pad(4096)
31
32      # The equivalent C structure looks like the following:
33      #
34      # struct global_header {
35      #     uint64_t num_vcpus;
36      #     uint64_t vcpu_list_off;
37      #     uint64_t num_cpus;
38      #     uint64_t percpu_off[MAX_CPUS];
39      # } __attribute__((aligned(4096)));
40
41      return ret
```

LISTING D.2: Python function for packing a global header.

```
1  # Pack a per-CPU structure containing core-local data.
2  def pack_percpu(core, vcpu_list, l1_slots, l1_slices,
3                  l1_slice_len, l1_table_len):
4
5      # Pack the slots and slices for this core
6      packed_l1_slots = pack_slots(l1_slots[core], vcpu_list)
7      packed_l1_slices = pack_slices_percpu(l1_slices[core], l1_slots)
```

```
8
9       # Generate and pack the per-CPU VCPU info list
10      percpu_vlist = [x for x in vcpu_list if core in x.cores]
11      packed_vinfo = pack_vcpu_info(percpu_vlist)
12
13      # Pack a header containing the offsets to each part of the per-CPU
14      # data structure. (1) The slots table, (2) the slice table, and
15      # (3) the local vCPU info table.
16      offset_slots = 4096
17      offset_slices = 4096 + packed_l1_slots.length()
18      offset_vcpus = offset_slices + packed_l1_slices.length()
19      packed_header = pack_percpu_header(
20                          nslots=len(l1_slots[core]),
21                          nslices=len(l1_slices[core]),
22                          slot_off=offset_slots,
23                          slice_off=offset_slices,
24                          slice_len=l1_slice_len,
25                          table_len=l1_table_len,
26                          nvcpus=len(percpu_vlist),
27                          vcpu_list_off=offset_vcpus
28                      )
29
30      # Pack the structure. Header, followed by slot table, followed
31      # by slice table, finally followed by local vCPU info structure.
32      ret = Pack()
33
34      ret.push_packed('header', packed_header)
35      ret.push_packed('slots', packed_l1_slots)
36      ret.push_packed('slices', packed_l1_slices)
37      ret.push_packed('vinfo', packed_vinfo)
38
39      return ret
```

LISTING D.3: Python function for packing a per-CPU structure.

```
1   # Pack a per-CPU header.
2   def pack_percpu_header(nslots, nslices, slot_off, slice_off, slice_len,
3                          table_len, nvcpus, vcpu_list_off):
```

```
4       ret = Pack()
5
6       # The number of slot structures in the slot list.
7       ret.push_u64('nslots', nslots)
8       # The offset of the slot list.
9       ret.push_u64('slot_list_off', slot_off)
10
11      # The number of slices in the slice table.
12      ret.push_u64('nslices', nslices)
13      # The offset of the slice table.
14      ret.push_u64('slice_list_off', slice_off)
15
16      # The time-interval of each slice.
17      ret.push_u64('slice_length', slice_len)
18      # The time-interval of the table itself.
19      ret.push_u64('table_length', table_len)
20
21      # The number of vCPU structures local to this CPU.
22      ret.push_u64('nvcpus', nvcpus)
23      # The offset of the vCPU info structure.
24      ret.push_u64('vcpu_list_off', vcpu_list_off)
25
26      # Pad everything to 4KP (page size)
27      ret.align_pad(4096)
28
29      return ret
```

LISTING D.4: Python function for packing the header for a per-CPU structure.

```
1  # Pack all the slices for the per-CPU structure
2  def pack_slices_percpu(slices, slots):
3      ret = Pack()
4
5      # Iterate over each slice, pack it, and append it.
6      for s in slices:
7          ret.push_packed('slice', pack_slice(s, slots))
8
9      # Align to page size (4KB)
```

```
10      ret.align_pad(4096)
11
12      return ret
```

LISTING D.5: Python function for packing the slice table for a particular CPU.

```
1  # Pack a single slice into a 64-byte structure.
2  def pack_slice(s, slots):
3      ret = Pack()
4
5      # The start and end time of this slice
6      ret.push_u64('start', s.t_from)
7      ret.push_u64('end', s.t_to)
8
9      # Pack the left slot (should always be set)
10     assert s.left_alloc != None
11     ret.push_u64('left', sched.sid_to_slot(s.left_alloc, slots))
12
13     # Pack middle slot (may be NULL; that is idle)
14     if s.idle_middle == None:
15         ret.push_u64('middle', 32767)
16     else:
17         ret.push_u64('middle', sched.sid_to_slot(s.idle_middle, slots))
18
19     # Pack the right slot (may be NULL; that is idle)
20     if s.right_alloc == None:
21         ret.push_u64('right', 32767)
22     else:
23         ret.push_u64('right', sched.sid_to_slot(s.right_alloc, slots))
24
25     # Align to 64 bytes.
26     ret.align_pad(64)
27
28     return ret
```

LISTING D.6: Python function for packing a single slice within the slice table.

```
1  # Pack all the slots for the per-CPU structure
```

```
2  def pack_slots(slots, vinfo):
3      ret = Pack()
4
5      # Iterate over each slot, pack it, and append it.
6      for s in slots:
7          ret.push_packed('slot', pack_slot(s, vinfo))
8
9      # Align to page size (4KB)
10     ret.align_pad(4096)
11
12     return ret
```

LISTING D.7: Python function for packing the slots for a particular CPU.

```
1  # Pack a single slot into a 64-byte structure.
2  def pack_slot(slot, vinfo):
3      ret = Pack()
4
5      # Offset of this slot's vCPU in global VCPU list
6      ret.push_u64('offset', slot.t_from)
7      # A blank pointer that the hypervisor overwrites
8      # to point to the real vCPU structure at runtime.
9      ret.push_u64('vcpu_ptr', sched.find_in_vinfo(vinfo, slot.dom_id, slot.vcpu_id)
10     # The time-interval length of this slot
11     ret.push_u64('length', slot.t_to - slot.t_from)
12     # The start and end times
13     ret.push_u64('start', slot.t_from)
14     ret.push_u64('end', slot.t_to)
15     # Padded to 64 bytes (cache-line size)
16     ret.align_pad(64)
17
18     return ret
```

LISTING D.8: Python function for packing a single slot within the scheduling table

```
1  # Pack all the vCPU structures for the each CPU
2  def pack_vcpu_info(vcpus):
3      ret = Pack()
```

```
4
5       # Iterate over each local vCPU, pack it, and append it.
6       for v in vcpus:
7           ret.push_packed('vcpu', pack_vcpu(v))
8
9       # Align to page size (4KB)
10      ret.align_pad(4096)
11
12      return ret
```

LISTING D.9: Python function for packing the vCPU information list.

```
1   # Pack a single VCPU info structure into a 64-byte image.
2   def pack_vcpu(v):
3       ret = Pack()
4
5       # The domain and vCPU ID
6       ret.push_u32('dom_id', int(v.dom))
7       ret.push_u32('vcpu_id', int(v.vcpu))
8
9       # A bitmask of CPUs this vCPU is scheduled on
10      cpumask = 0
11      for c in v.cores:
12          cpumask |= 1 << c
13      ret.push_u64('cpumask', cpumask)    # Pack the CPU mask (64 bits)
14
15      # A flag specifying if it's semi-partitioned.
16      if len(v.cores) > 1 or not v.burstable:
17          ret.push_u64('flags', 1)
18      else:
19          ret.push_u64('flags', 0)
20
21      # A blank pointer for storing address of the
22      # actual vCPU structure at runtime.
23      ret.push_u64('vcpu_ptr', 0)
24
25      # Align to 64 bytes
26      ret.align_pad(64)
```

```
27
28      return ret
```

LISTING D.10: Python function for packing the data for a single vCPU.

BIBLIOGRAPHY

[1] SchedCAT: Schedulability test collection and toolkit. https://www.mpi-sws.org/~bbb/projects/schedcat, 2018.

[2] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2, 2018.

[3] VMware ESXi: The purpose-built bare metal hypervisor. https://www.vmware.com/products/esxi-and-esx.html, 2019. Online; accessed 21 December 2019.

[4] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, et al. Impact of resource sharing on performance and performance prediction: A survey. In *International Conference on Concurrency Theory*, 2013.

[5] L. Abeni and D. Faggioli. An experimental analysis of the xen and kvm latencies. In *Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing*, 2019.

[6] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5), 2006.

[7] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[8] Amazon. Amazon Web Services - HIPAA compliance. https://aws.amazon.com/compliance/hipaa-compliance/.

[9] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[10] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Embedded and Real-Time Computing Systems and Applications*, 2006.

[11] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Proceedings of the Real-Time Systems Symposium*, 2008.

[12] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[13] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. 2007.

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, 1993.

[15] C. Bienia and K. Li. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[16] K. Bletsas and B. Andersson. Notional processors: An approach for multiprocessor scheduling. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[17] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *Real-Time Systems*, 47(4), 2011.

[18] B. Brandenburg and M. Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, 2016.

[19] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. *ACM SIGMETRICS Performance Evaluation Review*, 25(1), 1997.

[20] A. Burns, R. I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. *Real-Time Systems*, 48(1), 2012.

[21] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14(1), 2016.

[22] G. Casale, C. Ragusa, and P. Parpas. A feasibility study of host-level contention detection by guest virtual machines. In *IEEE 5th International Conference on Cloud Computing Technology and Science*, 2013.

[23] F. Cerqueira and B. Brandenburg. A comparison of scheduling latency in linux, PREEMPT-RT, and LITMUS^{RT}. In *Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2013.

[24] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. Hierarchical multiprocessor CPU reservations for the Linux kernel. In *Proceedings of the 5th international workshop on operating systems platforms for embedded real-time applications*, 2009.

[25] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Performance Evaluation Review*, 35(2), 2007.

[26] S. N. T. Chiueh and S. Brook. A survey on virtualization technologies. *RPE Report*, 142, 2005.

[27] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. *Approximation algorithms for NP-hard problems*, pages 46–93, 1996.

[28] W. E. Cohen. Tuning programs with OProfile. *Wide Open Magazine*, 1, 2004.

[29] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *The 26th ACM Symposium on Operating Systems Principles*, 2017.

[30] A. Crespo, I. Ripoll, and M. Masmano. Partitioned embedded architecture based on hypervisor: The XtratuM approach. In *European Dependable Computing Conference*, 2010.

[31] T. Cucinotta, G. Anastasi, and L. Abeni. Real-time virtual machines. In *Proceedings of the 29th IEEE Real-Time System Symposium Work-in-Progress Session*, 2008.

[32] T. Cucinotta, G. Anastasi, and L. Abeni. Respecting temporal constraints in virtualised services. In *33rd Annual IEEE International Computer Software and Applications Conference*, 2009.

[33] M. Danish, Y. Li, and R. West. Virtual-CPU scheduling in the quest operating system. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.

[34] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4), 2011.

[35] A. C. De Melo. The new Linux perf tools. In *Linux Kongress*, 2010.

[36] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2), 2013.

[37] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1), 1978.

[38] Y. Dong, Z. Yu, and G. Rose. SR-IOV networking in Xen: Architecture, design and implementation. In *Proceedings of the Workshop on I/O Virtualization*, 2008.

[39] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, 2001.

[40] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[41] F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, 2010.

[42] L. Furano, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-driven LLC allocation. In *Proceedings of the USENIX Annual Technical Conference*, 2016.

[43] M. García-Valls, T. Cucinotta, and C. Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9), 2014.

[44] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, volume 29. W. H. Freeman and Company, 2002.

[45] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual Execution Environments*, 2007.

[46] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM Computer Communication Review*, 39(1), 2008.

[47] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can JUMP them! In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[48] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2006.

[49] P. henning Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000.

[50] P. Holman and J. H. Anderson. Adapting Pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4), 2005.

[51] Intel. Intel processor counter monitor. `https://github.com/opcm/pcm`.

[52] Intel. Intel® 64 and IA-32 architectures software developer's manual: System programming guide. Technical report, Intel, 2011.

[53] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011.

[54] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message completion time in the cloud. Technical Report MSR-TR-2013-95, 2013. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=201418`.

[55] S. A. Javadi and A. Gandhi. DIAL: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing. In *IEEE International Conference on Autonomic Computing*, 2017.

[56] S. A. Javadi, S. Mehra, B. K. R. Vangoor, and A. Gandhi. UIE: User-centric interference estimation for cloud applications. In *IEEE International Conference on Cloud Engineering*, 2016.

[57] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical network performance isolation at the edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[58] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[59] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proceedings of the 8th ACM International Conference on Embedded Software*, 2008.

[60] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009.

[61] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2009.

[62] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, 2009.

[63] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv: Optimizing the operating system for virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2014.

[64] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *IEEE International Symposium on Performance Analysis of Systems & Software*, 2007.

[65] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1), 2003.

[66] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the Xen hypervisor. In *ACM SIGPLAN Notices*, volume 45, 2010.

[67] P. Leitner and J. Cito. Patterns in the chaos — a study of performance variation and predictability in public IaaS clouds. *ACM Transactions on Internet Technology*, 16(3), 2016.

[68] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, 2010.

[69] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[70] B. Lin and P. A. Dinda. VSched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2005.

[71] M. Lindberg. A survey of reservation-based scheduling. Technical Report TFRT-7618, Department of Automatic Control, Lund Institute of Technology, Lund University, 2007.

[72] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[73] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[74] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: a decade of wasted cores. In *Proceedings of the 11th European Conference on Computer Systems*, 2016.

[75] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for

the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[76] A. Madhavapeddy, D. J. Scott, J. Lango, M. Cavage, P. Helland, and D. Owens. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1), 2014.

[77] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-in-time summoning of Unikernels. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, 2015.

[78] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*, 2014.

[79] A. K. Maji, S. Mitra, and S. Bagchi. ICE: An integrated configuration engine for interference mitigation in cloud services. In *IEEE International Conference on Autonomic Computing*, 2015.

[80] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[81] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[82] A. Masrur, S. Drossler, T. Pfeuffer, and S. Chakraborty. VM-based real-time services for automotive control applications. In *IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2010.

[83] E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt. Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, 2014.

[84] B. McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[85] R. McDougall and J. Mauro. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Pearson Education, 2006.

[86] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In

Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, 2005.

[87] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, (239), 2014.

[88] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP users group conference*, 1999.

[89] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, New York, NY, USA, 2010.

[90] R. Nikolaev and G. Back. Perfctr-Xen: a framework for performance counter virtualization. In *ACM SIGPLAN Notices*, volume 46, 2011.

[91] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[92] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[93] C. S. Pabla. Completely fair scheduler. *Linux Journal*, 2009(184), 2009.

[94] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the ACM Conference on SIGCOMM*, 2014.

[95] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7), 1974.

[96] D. Price and A. Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX Conference on System Administration*, 2004.

[97] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Proceedings of the IEEE 32nd Real-Time Systems Symposium*, 2011.

[98] J. Reinders. VTune performance analyzer essentials. *Intel Press*, 2005.

[99] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the SIGOPS European Conference on Computer Systems*, 2013.

[100] O. Sefraoui, M. Aissaoui, and M. Eleuldj. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[101] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[102] A. Srinivasan and J. H. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002.

[103] O. Sukwong, A. Sangpetch, and H. S. Kim. SageShift: Managing SLAs for highly consolidated cloud. In *Proceedings of the 31st Annual IEEE International Conference on Computer Communications*, 2012.

[104] G. Taylor, P. Davies, and M. Farmwald. The TLB slice—a low-cost high-speed address translation mechanism. In *Proceedings of the 17th annual International Symposium on Computer Architecture*, 1990.

[105] G. Tene. How not to measure latency. *Low Latency Summit*, page 68, 2013.

[106] US Air Force. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the USENIX Security Symposium*, 2000.

[107] K. van Surksum. The CPU scheduler in VMware vSphere 5.1. Technical report, VMware, 2013.

[108] M. Vanga, A. Gujarati, and B. Brandenburg. Tableau: a high-throughput and predictable vm scheduler for high-density workloads. In *Proceedings of the EuroSys Conference*, 2018.

[109] N. Vasiundefined, D. Novakoviundefined, S. Miučin, D. Kostiundefined, and R. Bianchini. DejaVu: Accelerating resource allocation in virtualized environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2012.

[110] S. Wang, W. Zhang, T. Wang, C. Ye, and T. Huang. VMon: Monitoring and quantifying virtual machine interference via hardware performance counter. In *IEEE 39th Annual Computer Software and Applications Conference*, volume 2, 2015.

[111] A. Waterland. stress POSIX workload generator. `http://people.seas.harvard.edu/~apw/stress`, 2013.

[112] J. Whiteaker, F. Schneider, and R. Teixeira. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1), 2011.

[113] T. Willhalm, R. Dementiev, and P. Fay. Intel performance counter monitor - a better way to measure CPU utilization. Technical report, Intel, 2012.

[114] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011.

[115] C. Wong, I. Tan, R. Kumari, J. Lam, and W. Fun. Fairness and interactive performance of o (1) and CFS linux kernel schedulers. In *Proceedings of the International Symposium on Information Technology*, 2008.

[116] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey. Towards achieving fairness in the Linux scheduler. *ACM SIGOPS Operating Systems Review*, 42(5), 2008.

[117] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11^{th} USENIX Security Symposium*, 2002.

[118] J. Wu and J.-F. Li. ERTDS: A dynamic CPU scheduler for xen virtualization systems. In *International Conference on Applied System Innovation*, 2017.

[119] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 21^{st} Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[120] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *Proceedings of the International Conference on Embedded Software*, 2011.

[121] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multicore virtual machine scheduling in Xen. In *International Conference on Embedded Software*, 2014.

[122] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multicore virtual machine scheduling in Xen. In *Proceedings of the 14^{th} International Conference on Embedded Software*, 2014.

[123] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, 2017.

[124] T. Xu, X. Sui, Z. Yao, J. Ma, Y. Bao, and L. Zhang. Rethinking virtual machine interference in the era of cloud applications. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications & IEEE International Conference on Embedded and Ubiquitous Computing*, 2013.

[125] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[126] P. Yu, M. Xia, Q. Lin, M. Zhu, S. Gao, Z. Qi, K. Chen, and H. Guan. Real-time enhancement for Xen hypervisor. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, 2010.

[127] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2013.

[128] W. Zhang, S. Rajasekaran, and T. Wood. Big data in the background: Maximizing productivity while minimizing virtual machine interference. In *Proceedings of the Workshop on Architectures and Systems for Big Data*, 2013.

[129] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009.

[130] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.