SCALING SYNCHRONIZATION PRIMITIVES

A Dissertation
Presented to
The Academic Faculty

By

Sanidhya Kashyap

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2020

Copyright © Sanidhya Kashyap 2020

SCALING SYNCHRONIZATION PRIMITIVES

Approved by:

Dr. Taesoo Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Changwoo Min, Co-advisor
The Bradley Department of Electrical and Computer Engineering
Virginia Tech

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Dr. Irina Calciu
VMWare Research

Dr. Joy Arulraj
School of Computer Science
Georgia Institute of Technology

Date Approved: June 11, 2020

To *maa*: for her strength, wisdom, and love.

To *papa*: for his unconditional belief in me.

To my *brother*: for being my critic.

To my background thread.

ACKNOWLEDGEMENTS

I am fortunate to have had the opportunity to collaborate with and learn from amazing people. First, I owe my deepest gratitude to my advisor, Taesoo Kim. Throughout my Ph.D., Taesoo gave me an ample amount of freedom to explore ideas and collaborate with anyone. I am never going to forget these words: "Do whatever you want to do." His in-depth technical insight, concrete feedback, advice, encouragement, and guidance has led me to where I am today. I hope one day, I can be a good advisor to my future students, as Taesoo has been to me.

Another person who had an immeasurable impact on my life is my co-advisor: Changwoo Min. His vast knowledge of designing parallel systems was essential to realize any of this work. I am never going to forget our discussions, which I still miss. I used to bug him almost every day during his postdoc days, and I still do when I am excited to share some new ideas. I am immensely grateful to him, as he was my mentor, friend, and guide, all at the same time, and continues to do so.

In my later part of Ph.D., I was fortunate to work with Irina Calciu. Working with Irina always kept me on edge, as she asked too many (difficult) questions to make me rethink about the work. I hope this continues for the foreseeable future. I also want to thank the other members of my committee: Ada Gavrilovska, and Joy Arulraj. Both of them were always available to discuss ideas and even my career options.

I want to thank all the amazing folks that I got to know, learn from, and work with through these years, notably: Sudarsun Kannan, Virendra Maratha, Seulbae Kim, Meng Xu, Insu Yun, Mingwei Shih, Steffen Maass, Mohan Kumar, Tushar Krishna, Fan Sang, Ren Ding, Kyuhong Park, Pradeep Fernando, Mansour Alharthi, Hong Hu, Woonhak Kang, Byoungyoung Lee, Chengyu Song, Woonhak Kang, Wen Xu, Kangnyeon Kim, Hyungon Moon, Seulbae Kim, Meng, Xu, Jean Pierre Lozi, Margo Seltzer, Alex Kogan, Dave Dice, Hong Hu, Hanqing Zhao, Se Kwon Lee, Soujanya Ponnapalli, Madhavan Krishnan

Ramanathan, Sujin Park, Chulwon Kang, and Xiaohe Cheng. I am really grateful to Seulbae for being a good friend. I want to thank our previous Ph.D. coordinator, Venkat, for his help in navigating various PhD-related administrative issues and even suggesting I work with Taesoo. I am also grateful to our administrative problem-solvers, who made my life easier, especially Elizabeth Ndongi, Trinh Doan, and Sue Jean Chae.

Finally, I want to thank my parents for their support and patience along this journey. I want to thank my friend, Jaspal, who has been there through thick and thin, remotely. I am especially grateful to my brother, from whom I have a lot to learn in the arena of life, philosophy, and about myself.

TABLE OF CONTENTS

Acknowledgments iv

List of Tables x

List of Figures xii

List of Pseudo-Code xvii

Summary xviii

Chapter 1: Introduction 1

1.1 Ordering in Concurrency Control Algorithms 3

1.2 Double Scheduling in Virtualization 4

1.3 Scalable and Practical Locking Primitives 4

1.4 Outline and Contributions 5

Chapter 2: Background and Motivation 6

2.1 A Primer on Multicore Machines 6

2.2 Ordering in Concurrency 7

2.2.1 Multicore Hardware Clocks 8

2.3 Locking Primitives 9

2.3.1 Evolution of Lock design 10

2.3.2 Locks in the kernel space (Linux) 12

2.3.3 Locking Bottlenecks in Deployed File Systems 13

2.4 Double Scheduling in VMs 15

2.5 Conclusion 17

Chapter 3: Ordering Primitive 18

3.1 Ordo: A Scalable Ordering Primitive 19

3.1.1 Embracing Uncertainty in Clock: Ordo API 20

3.1.2 Measuring Uncertainty between Clocks: Calculating ORDO_BOUNDARY 20

3.2 Algorithms with Ordo without Uncertainty 26

3.2.1 Read-Log-Update (RLU) 26

3.2.2 Concurrency Control for Databases 29

3.2.3 Software Transactional Memory (TL2) 31

3.2.4 Oplog: An Update-heavy Data Structures Library 33

3.3 Implementation 33

3.4 Evaluation 34

3.4.1 Scalability of Invariant Hardware Clocks 36

3.4.2 Evaluating Ordo Primitive 36

3.4.3 Physical Timestamping: Oplog 38

3.4.4 Read Log Update 39

3.4.5 Concurrency Control Mechanism 43

3.4.6 Software Transactional Memory 46

3.4.7 Sensitivity Analysis of ORDO_BOUNDARY 48

3.5 Chapter Summary 49

Chapter 4: Impact of Scheduling on Virtual Machines 50

4.1 Design 51

4.1.1 Lightweight Para-virtualized methods 53

4.1.2 Eventual Fairness with Selective Scheduling 55

4.2 Use Case 56

4.3 Implementation 57

4.4 Evaluation 59

4.4.1 Overhead of *e*CS 60

4.4.2 Performance in an Over-committed Scenario 61

4.4.3 Performance in an Under-committed Case 67

4.4.4 Addressing BWW Problem via *e*CS 69

4.4.5 System Eventual Fairness 70

4.5 Chapter Summary 72

Chapter 5: Scalable Locking Primitives 73

5.1 Dominating Factors in Lock Design 74

5.2 SHFLLOCKS 77

5.2.1 The Shuffling Mechanism 78

5.2.2 SHFLLOCKS Design 78

5.3 Implementation 90

5.4 Evaluation 90

5.4.1 SHFLLOCK Performance Comparison 91

5.4.2 Improving Application Performance 95

5.4.3 Performance Breakdown 99

5.4.4 Performance With Userspace SHFLLOCK 101

5.5 Chapter Summary 105

Chapter 6: Related Work 106

6.1 Ordering in Concurrency Control Algorithms 106

6.2 Double Scheduling in VMs 108

6.3 Locking Primitives 110

Chapter 7: Reflections 112

7.1 Limitations 112

7.1.1 Ordering in Concurrency with `Ordo` 112

7.1.2 Enlightened Critical Sections 113

7.1.3 Shuffling-based Lock Algorithms 114

7.2 Future Work 115

Chapter 8: Conclusion 117

References 135

LIST OF TABLES

2.1 Evolution of synchronization primitives in the last 15 years in Linux along with their introduction with the corresponding Linux version. 11

2.2 Identified lock-based scalability bottlenecks in tested file systems with `FxMark` [77]. 14

3.1 Various machine configurations that we use in our evaluation as well as the calculated offset between cores. While min is the minimum offset between cores, max is the global offset, called `ORDO_BOUNDARY` (refer to Figure 3.1), which we used, including up to the maximum hardware threads (Cores$*$SMT) in a machine. 36

4.1 Set of para-virtualized methods exposed by the hypervisor to a VM for providing hints to the hypervisor to mitigate double scheduling. These methods provide hints to the hypervisor and VM via shared memory. A `vCPU` relies on the first four methods to ask for an extra schedule to overcome `LHP`, `LWP`, `RP`, `RRP`, and `ICP`. Meanwhile, a `vCPU` gets hints from the hypervisor by using the last two methods to mitigate `LWP` and `BWW` problems. The `cpu_id` is the core id that is used by tasks running inside a guest OS. †Currently, `is_vcpu_preempted()` is already exposed to the VM in Linux. 52

4.2 Applicability of our six lightweight para-virtualized methods that strive to address the symptoms of double scheduling. 56

4.3 e`CS` requires small modifications to the existing Linux kernel, and the annotation effort is also minimal: 60 LoC changes to support the 10 million LoC Linux kernel that has around 12,000 of lock instances with 85,000 lock invocations. 58

4.4 Cost of using our lightweight para-virtualized methods with various synchronization primitives and mechanism. *1 core* and *80 core* denote the time (in ns) to execute an empty critical section with one and 80 threads, respectively. Although, our approach slightly adds an overhead on a single core count, there is no performance degradation for our evaluated workloads. 60

5.1 Dominant factors affecting locks that are in use in the Linux kernel or are the state-of-the-art for NUMA architecture. Cache-line movement refers to shfllock/data movement inside a critical section. Boxes represent the scalability of locks with increasing thread count from one thread to threads within a socket to all threads among multiple sockets. Core subscription is only applicable to blocking locks and denotes the best throughput for a varying number of threads. Both `mutex` and `CST` are sub-optimal when under-subscribed but maintain good throughput once they are over-subscribed. Memory footprint is the memory allocation for locks: the size of each lock instance (per lock), a queue node required by each waiting thread before entering the critical section (per waiter), and a queue node retained by a lock holder within the critical section (per lock-holder). If the lock holder uses the queue node, which happens for MCS, CNA, and Cohort locks, the thread must keep track of the node, as it can acquire multiple locks: a common scenario in Linux. Note that queue nodes can be allocated on the stack for each algorithm. However, in practice, a lock user needs to explicitly allocate it on the stack for MCS, CNA, and Cohort locks, while `mutex`, `CST`, and SHFLLOCKS avoid this complexity. We also summarize the number of atomic instructions in the non-contended/contended scenarios. 75

5.2 Locks evaluated in both the kernel space and the userspace. In the kernel space, we replace all locks with SHFLLOCKS. We use `LD_PRELOAD` to replace all the mutex-based locks in the userspace. 91

5.3 Lock usage in various micro-benchmarks [77, 146]. 92

LIST OF FIGURES

2.1 Multicore machine design. 6

2.2 Indirect metric of the growing complexity of lock usage: the number of lock() API calls in the Linux kernel source code. 10

3.1 Algorithm to calculate the ORDO_BOUNDARY: a system-wide global offset. . . 21

3.2 Calculating offset ($\delta_{i \leftrightarrow j}$) using pairwise one-way-delay latency between clocks ($\mathtt{c}_i$ and $\mathtt{c}_j$). Δ_{ij} and Δ_{ji} are the physical offsets between $\mathtt{c}_i$ and $\mathtt{c}_j$, and $\mathtt{c}_j$ and $\mathtt{c}_i$, respectively. δ_{ij} and δ_{ji} are measured offsets with our approach. Depending on physical and measured offsets, there are four possible cases. Unlike existing clock-synchronization protocols [56, 57, 54, 95][3] that average the calculated latency based on RTT, we consider each direction separately to measure the offset and consider only the direction that is always greater the positive physical offset, such as cases 1 and 4. . . 23

3.3 Micro evaluation of the invariant hardware clocks used by the Ordo primitive. (a) shows the cost of a single timestamping instruction when it is executed by varying the number of threads in parallel; (b) shows the number of per-core generated timestamps in a micro second with atomic increments (**A**) and with new_time() (**O**), which generates timestamps at each ORDO_BOUNDARY. 35

3.4 Clock offsets for all pairs of cores. The measured offset varies from a minimum of 70 ns to 1,100 ns for all architectures. Both Xeon (a) and ARM (d) machines show that the one of the sockets has a 4–8× higher offset than the others. To confirm this, we measured the bandwidth between the sockets, which is symmetric in both machines. 37

3.5 Throughput of Exim mail-server on a 240-core machine. Vanilla represents the unmodified Linux kernel, while Oplog is the rmap ordo/data structure modified by the Oplog API in the Linux kernel. Oplog[ORDO] is the extension of Oplog with the Ordo primitive. 39

3.6 Throughput of the hash table with RLU and RLU^ORDO for various update ratios of 2% and 40% updates. The user space hash table has 1,000 buckets with 100 nodes. We experiment it on four machines from Table 3.1. 40

3.7 Throughput of the citrus tree with RLU and RLU^ORDO that consists of 100,000 nodes with varying updates ratio of 10% and 40% on various machines. . . 41

3.8 Throughput of the hash table benchmark (40% updates) with the deferred-based RLU and RLU^ORDO approach on the Xeon machine. Even with the deferred-based approach, the cost of the global clock is still visible after crossing the NUMA boundary. 43

3.9 Throughput of various concurrency control algorithms of ordo/databases for read-only transactions (100% reads) using the YCSB benchmark on various architectures. We modify the existing OCC and Hekaton algorithms to use our Ordo primitive (OCC^ORDO and Hekaton^ORDO, respectively) and compare them against the state-of-the-art OCC algorithms: Silo and TicToc. Our modifications removes the logical clock bottleneck across various architectures. 44

3.10 Throughput and abort rates of concurrency control algorithms for TPC-C benchmark with 60 warehouses on 240 Intel Xeon machine. 45

3.11 Speedup of the STAMP benchmark with respect to the sequential execution on Xeon machine for TL2 and TL2^ORDO algorithms. TL2^ORDO improves the throughput up to 3.8× by alleviating the cache-line contention that occurs of the global logical clock. TL2^ORDO shows significant improvement in the case of workloads running with very short transactions (Kmeans and Ssca2) and the ones with very long transactions by decreasing their aborts due to the cache-line mitigation (Labyrinth). 47

3.12 Normalized throughput of the RLU^ORDO algorithm for varying ORDO_BOUNDARY on the Xeon machine on 1-core, 1-socket (30 cores), and 8-sockets (240 cores) with 98% reads and 2% writes. We vary the ORDO_BOUNDARY from 1/8×–8× for all three configurations, which shows that the throughput varies by only $\pm 3\%$, thereby proving two points: 1) timestamping is one of the bottlenecks, and 2) ORDO_BOUNDARY does not act as a backoff mechanism for such logical timestamping-based algorithms. 48

4.1 Overview of the information flow between a VM and a hypervisor. Each vCPU has a per-CPU state that is shared with the hypervisor, denoted as eCS state. Figure (a) shows how the $vCPU_2$ relays information about an eCS to the hypervisor. On entering a critical section or an interrupt context (❶), $vCPU_2$ updates the non_preemptable_ecs_count (❷). After a while, before scheduling out $vCPU_2$, the hypervisor reads its eCS state (❸), and allows it run for one more schedule to mitigate any of the double scheduling problems. Figure (b) shows how the hypervisor shares the information whether a vCPU is preempted or a physical CPU is overloaded, at the schedule boundary. For instance, the hypervisor marks vcpu_preempted, while scheduling out a vCPU; or updates pcpu_overloaded flag to one if the number of active tasks on that physical CPU is more than one. Both try to further mitigate LWP and BWW problems. 54

4.2 Analysis of real-world workloads (Apache and Psearchy) in an over-committed scenario, *i.e.*, two instances of VM are executing the same workload. Column (i) represents the scalability of selected workloads in three settings: PVM, HVM, and with eCS annotations. Column (ii) represents the number of preemptions caught and prevented by the hypervisor with our methods. Column (iii) represents the type of preemptions caught by the hypervisor (refer Table 4.4). By allowing an extra schedule, our approach reduces preemptions by 85–100% and improve scalability of applications by up to 2.5×, while observing almost all types of preemptions for each workload. 62

4.3 Analysis of real-world workloads (Metis and Pbzip2) in an over-committed scenario, *i.e.*, two instances of VM are executing the same workload. Column (i) represents the scalability of selected workloads in three settings: PVM, HVM, and with eCS annotations. Column (ii) represents the number of preemptions caught and prevented by the hypervisor with our methods. Column (iii) represents the type of preemptions caught by the hypervisor (refer Table 4.4). By allowing an extra schedule, our approach reduces preemptions by 85–100% and improve scalability of applications by up to 2.5×, while observing almost all types of preemptions for each workload. 63

4.4 CDF of the latency of requests for the Apache web server workload in both under- and over-committed scenarios at 80 cores. It clearly shows the impact of eCS in the over-committed scenario, while having minimal impact in the under-committed case. 65

4.5 Performance of real-world workloads when running on the bare metal (Host), and inside a VM with three configurations: PVM, HVM, and with eCS annotations. In this scenario, only one VM is running. We use Host as the baseline for the comparison because we consider Host to have almost optimal performance. 68

4.6 Impact of both BWW problem and eCS method (refer Hypersivor → VM in Table 4.1) on Psearchy in both under- and over-committed scenarios. . . . 69

4.7 Fairness in eCS. Running time of a vCPU of two co-scheduled VMs (VM_1 and VM_2) with eCS annotations for a period of 10 seconds with 100 ms window granularity while executing a kernel intensive task (reading the contents of a file) that involves read side of rwsem. (a) shows the difference in running time of vCPU per window granularity as well as the number of preemptions occurring per window, while (b) illustrates the cumulative running time, and shows that the hypervisor maintains eventual fairness in the system, even if VM_2 is allowed extra schedules. Both VMs get 4.95 seconds to run. 71

5.1 SHFLLOCK^{NB} example. The lock structure consists of a state (glock) and the queue tail. The first byte of glock is the lock/unlock state, while the second byte denotes whether stealing is allowed. We encode multiple information in the qnode structure. (a) Initially, there is no lock holder. (b) t_0 successfully acquires the lock via CAS and enters the critical section. (c) t_1, of socket 1, executes SWAP on the lock's tail after the CAS failure on TAS. (d) Similarly, t_2 from socket 1, also joins the queue. (e) Now, there are five waiters (t_1–t_5) waiting for the lock. t_1 is the very first waiter, so it becomes the shuffler and traverses the queue to find waiters from the same socket. t_1 then moves t_4 (same socket) after t_2. (f) After the traversal, t_1 selects t_4 as the next shuffler. (g) t_4 acquires the lock after t_1 and t_2 have executed their critical sections. At this point, t_3 becomes the shuffler. 81

5.2 A running example of how a shuffler shuffles waiters with the same socket ID and wakes them up. (a) t_0 is the lock holder; t_1 is the shuffler and is traversing the queue. t_2 is sleeping, but t_1 wakes it up. (b) t_2 becomes active, while t_1 continues shuffling and reaches t_4, t_1 first moves t_4 after t_2, and wakes up t_4 to mitigate the wakeup latency. (c) When t_0 releases the lock, t_1 acquires it; t_2 and t_4 are actively spinning for their turn; t_4 is the shuffler. 86

5.3 Impact of non-blocking locks on the scalability of micro-benchmarks [77, 146]. Refer to Table 5.3 for lock usage. Here, Stock refers to the default spinlock. 92

5.4 Impact of blocking locks on the scalability of micro-benchmarks with up to 2× over-subscription (384 threads: we pin two threads on each core). Cohort and CST are the non-blocking and blocking hierarchical locks, respectively. Refer to Table 5.3 for lock usage. 93

5.5 Impact of locks on application scalability and on memory footprint, while running three applications with SHFLLOCKS, Linux stock version (Stock), CNA, CST, and Cohort. Refer to Table 5.2 for specific changes. SHFLLOCK reduces the memory footprint because of the blocking locks that are embedded in inodes, task structure, and memory management structures. 96

5.6 Impact on throughput and long-term fairness of non-blocking and blocking locks on the hash-table benchmark. For blocking locks, we over-subscribe the system by 4×. We also include the factor analysis of several phases introduced by SHFLLOCK^{NB}, and the number of wakeups in the critical path for SHFLLOCK^{B}. Later, we show the impact on throughput with centralized readers-writer locks: Stock and SHFLLOCK^{RW} for 1% and 50% writes up to 4× over-subscription. 100

5.7 Total throughput of LevelDB benchmark and the streamcluster benchmark with various blocking and non-blocking locks. We further over-subscribe the cores for levelDB (b) to test the impact of blocking locks with 4× the number of cores. 102

5.8 Impact of locks and their memory allocation overhead on the scalability of Dedup. We report the overall memory allocation overhead that is used during the entire run, with respect to `pthread`. 104

LIST OF PSEUDO-CODE

3.1 `Ordo` clock API. The `get_time()` method returns the current timestamp without reordering instructions. 19

3.2 Pseudo-code of logical timestamping algorithms used in `STM` [1, 2] and databases [3, 4, 5], and that of physical timestamping used in `Oplog` [6]. 25

3.3 `RLU` pseudo-code including our changes. 27

5.1 Pseudo-code of the non-blocking version of SHFLLOCKS and the shuffling mechanism. 84

5.2 The extra modification required to convert our non-blocking version of SHFLLOCK to a blocking one. 87

5.3 An optimization for avoiding a waiter wakeup issue in the critical path with an extra state update before the `TAS` lock. 88

SUMMARY

Over the past decade, multicore machines have become the norm. A single machine is capable of having thousands of hardware threads or cores. Even cloud providers offer such large multicore machines for data processing engines and databases. Thus, a fundamental question arises is how efficient are existing *synchronization primitives*—timestamping and locking—that developers use for designing concurrent, scalable, and performant applications. This dissertation focuses on understanding the scalability aspect of these primitives, and presents new algorithms and approaches, that either leverage the hardware or the application domain knowledge, to scale up to hundreds of cores.

First, the thesis presents `Ordo`, a scalable ordering or timestamping primitive, that forms the basis of designing scalable timestamp-based concurrency control mechanisms. `Ordo` relies on *invariant hardware clocks* and provides a notion of a globally synchronized clock within a machine. We use the `Ordo` primitive to redesign a synchronization mechanism and concurrency control mechanisms in databases and software transactional memory.

Later, this thesis focuses on the scalability aspect of locks in both virtualized and non-virtualized scenarios. In a virtualized environment, we identify that these locks suffer from various preemption issues due to a semantic gap between the hypervisor shceduler and a virtual machine scheduler—the double scheduling problem. We address this problem by bridging this gap, in which both the hypervisor and virtual machines share minimal scheduling information to avoid the preemption problems.

Finally, we focus on the design of lock algorithms in general. We find that locks in practice have discrepancies from locks in design. For example, popular spinlocks suffer from excessive cache-line bouncing in multicore (NUMA) systems, while state-of-the-art locks exhibit sub-par single-thread performance. We classify several dominating factors that impact the performance of lock algorithms. We then propose a new technique, *shuffling*, that can dynamically accommodate all these factors, without slowing down the critical

path of the lock. The key idea of shuffling is to re-order the queue of threads waiting to acquire the lock with some pre-established policy. Using shuffling, we propose a family of locking algorithms, called SHFLLOCKS that respect all factors, efficiently utilize waiters, and achieve the best performance.

CHAPTER 1
INTRODUCTION

The last decade has seen dramatic changes in the hardware landscape. Until the last decade, processor vendors improved application performance by increasing CPU frequency. However, physical limitations in the form of CPU to heat up has resulted in pursuing the direction of multicore machines. As a result, microprocessor vendors have been building bigger multi-core and multi-socket (NUMA) machines [7, 8]. These machines provide a massive amount of memory, accessible by tens-to-hundreds of CPUs in a single machine. Besides, these machines are also capable of supporting non-volatile memory [9, 10, 11], storage devices [12, 13, 14], and even specialized support for hardware virtualization [15, 16, 17]. Because of fast-evolving hardware, concurrent programming is now the de-facto standard to design today's application to leverage today's hardware.

On the software front, almost every application, such as databases [18], processing engines [19, 20], and operating systems are now concurrent. Application developers are parallelizing their applications to use multiples of available cores efficiently. Also, to further improve hardware utilization, organizations are predominantly using virtualization to run multiples of applications together. This scheduling leads to over-subscribing hardware resources, even for large multicore and multi-socket machines.

Thus, application developers rely on various types of synchronization primitives to design concurrent data structures, algorithms, concurrency frameworks, and applications. These primitives handle the concurrency of operations by not only ensuring the correct use of the shared resources but also scheduling the concurrent events. For example, several applications use timestamping as an ordering mechanisms to design 1) concurrency control algorithms in databases and software transactional memory (`STM`); 2) logging in databases and file systems; and 3) memory reclamation in both memory allocators and garbage

collectors. Furthermore, application developers rely on the lock-based programming model to design concurrent and parallel applications. Lock-based programming model provides mutual exclusion (*i.e.*, exclusive access to shared resources) and also schedule the concurrent requests to access or modify the shared resource. The reason this model is so successful because it is the easiest to reason about correctness and enables easy composability of multiple data structures.

The basic premise of these primitives is to not only ensure application correctness but also have almost negligible overhead while scheduling concurrent events in an application. Unfortunately, with today's evolving hardware and ever-changing application requirements, most of these synchronization primitives have been designed specifically for either hardware or software requirements. For instance, first, most of the timestamp-based concurrency algorithms rely on atomic instructions that become scalability bottleneck with increasing thread count. Second, in the case of lock-based programming, there is no lock algorithm that satisfies various factors, such as data movement, thread contention, over-subscription, and memory footprint, which impact the scalability of locks and their adoption. Finally, the introduction of multiple layers of abstractions, such as virtualization, introduces scheduling overhead for VMs. This overhead stems from the missing semantic information across layers that inhibits the forward progress of applications running inside VMs.

Thesis Statement

Synchronization primitives are the basic building blocks for today's software stacks that not only ensure application correctness but also schedule concurrent events. Hence, this thesis further improves the performance of applications by providing right abstractions for these primitives that leverage both software and hardware interfaces to efficiently schedule such concurrent events.

This thesis focuses on efficiently scheduling concurrent events at various levels with respect to synchronization primitives: 1) leveraging hardware to minimize ordering over-

head (*timestamping*); 2) exposing semantic information across layers to solve the *double-scheduling* issues; and 3) decoupling the policy from the design/implementation of *lock* algorithms. Hence, this thesis makes the following contributions:

1.1 Ordering in Concurrency Control Algorithms

Chapter 3 describes `Ordo`, a scalable timestamping primitive for multicore machines that employs **invariant hardware clocks** to address problem of timestamping. This invariant hardware clock is already supported by current major processor architectures [21, 22, 23, 24]. An invariant clock has a unique property: It is monotonically increasing and has a constant skew, regardless of dynamic frequency and voltage scaling, and resets itself to zero whenever a machine resets (on receiving the RESET signal), which is ensured by the processor vendors [25, 26, 22, 24]. However, assuming that all invariant clocks in a machine are synchronized is incorrect because processors do not receive the RESET signal at the same time, which makes these clocks unusable. Thus, we cannot compare two clocks with confidence when correctly designing any time-based concurrent algorithms. The comparison of clocks with no confidence makes them unusable to correctly design a concurrent algorithm. To provide a guarantee of correct use of these clocks, we propose a new primitive, called `Ordo`, that embraces uncertainty while comparing two clocks and provides an illusion of a *globally synchronized hardware clock* in a single machine. We find that various timestamp-based algorithms can benefit from the `Ordo` primitive. We expose three simple clock-specific methods to replace existing clock with `Ordo` in applications. The only trick lies in handling the uncertainty window. We modify both physical and logical timestamp-based algorithms with the `Ordo` primitive. Some examples include concurrent data structure libraries and concurrency control algorithms for `STM` and databases.

1.2 Double Scheduling in Virtualization

Cloud providers provide the notion of horizontal as well as vertical application scaling. By running oversubscribing resources. Unfortunately, the multiplexing of VMs introduces the *double scheduling problem*: 1) the guest OS schedules processes on virtual CPUs (vCPUs) and 2) the hypervisor schedules vCPUs on physical CPUs. The root cause of this problem is a semantic gap between a hypervisor and guest OSes. Chapter 4 identifies several symptoms of the problem that have been individually studied and are very limited. We present an overall one-shot solution with a particular insight: if a certain component of a guest OS is allowed to proceed further, that guest OS will make forward progress. These critical components are the shared critical sections that synchronization mechanisms guard to ensure OS correctness. We bridge the semantic gap between a guest OS and the hypervisor with the following four ideas for making an effective scheduling decision to allow its forward progress. First, we consider shared resources running inside VMs as a critical component. Such resources are guarded by locks or are executing inside interrupt contexts Hence, these components act as forward progress indicators for an applications running inside VMs. Second, we devise a set of para-virtualized methods that annotate these critical components as *enlightened critical sections* (eCS). These methods are lightweight and rely on shared memory operations to notify a hypervisor from the VM and vice-versa, Third, the hypervisor can now figure out whether a vCPU is executing an eCS and can reschedule it. Finally, we leverage our methods to design a virtualized schedule-aware spinning strategy that allows VMs to be more cooperative to other VMs running on a machine.

1.3 Scalable and Practical Locking Primitives

Chapter 5 answers a fundamental question regarding the design of locking primitives. We first present four dominating factors that affect the locks' scalability and their adoption for evolving hardware and software requirements. Moreover, none of the existing lock

algorithms meet all the required criteria. We address all these factors by designing a new breed of locking protocols, called SHFLLOCKS, that are highly performant and practical locks. SHFLLOCKS rely on the *shuffling* technique, that allows the decoupling of the lock design from policy enforcement, such as NUMA-awareness and parking/wake-up strategies. Moreover, shuffling enables enforcing these policies mostly off the critical path by the waiters. Our evaluation in both kernel space and userspace shows that SHFLLOCKS maintain the best throughput regardless of the number of threads contending for the lock.

1.4 Outline and Contributions

The main contributions of this thesis are designing 1) a new ordering primitive that handles the scalability bottleneck in timestamp concurrency control algorithms, 2) understanding the scalability bottlenecks that arise due to synchronization primitives, 3) addressing the limitations of existing synchronization primitives by designing five new algorithms 4) even in the case of virtualized environments.

The rest of this dissertation is organized in the following chapters. Chapter 2 provides the necessary background used throughout this dissertation, such as the importance of ordering, synchronization in the age of many-core machines across the software stack. Chapter 3 presents the `Ordo`, a scalable ordering primitive for multicore machines. The later part of this dissertation focuses on rethinking the design of synchronization primitives and the associated problems occurring in a virtualized environment. Chapter 4 describes an approach to address the double scheduling problem in a virtualized environment. Chapter 5 presents a family of locking algorithms that use a new technique, called *shuffling*, that allows the decoupling of lock design from policy enforcement. Then Chapter 6 peruses over a set of related works that led us to this dissertation. In Chapter 7, we discuss some of the limitations of proposed approaches and some open-ended questions. Finally, Chapter 8 concludes this dissertation.

CHAPTER 2
BACKGROUND AND MOTIVATION

Synchronization primitives are the backbones for designing concurrent applications. At the hardware level, users rely on hardware constructs, such as atomic instructions that provide a linearizability guarantee [27]. At the software level, we rely on these atomic instructions to provide basic software abstractions, such as synchronization frameworks, and locking primitives for designing concurrent applications. This chapter offers some general ideas, concepts, and background material for such constructs on multicore machines.

2.1 A Primer on Multicore Machines

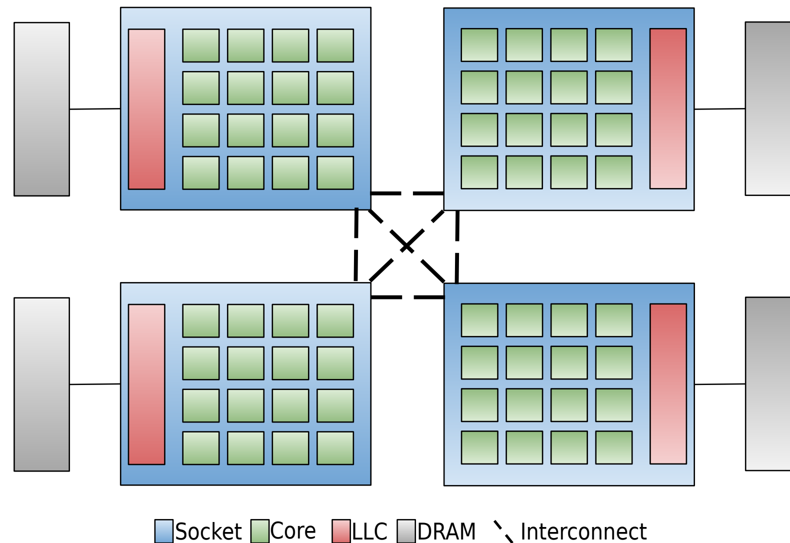

Figure 2.1: Multicore machine design.

Figure 2.1 shows a modern multicore machine, which comprises several sockets. Each socket has several homogeneous cores and a set of private and shared caches. Generally, a core has its private cache (*L1* and *L2*). It accesses a bigger shared cache (*LLC*) with

other cores in a socket. Moreover, each core stores data in DRAM, which is accessible via the memory controller. These multicore machines have non-uniform memory access (NUMA) architecture, *i.e.*, a high-speed interconnect (*e.g.*, Intel QPI/UPI [28, 29] or AMD HyperTransport [30]) links all sockets together. On such machines, accessing the local socket memory is faster than accessing the remote socket memory (hence, the term NUMA). Moreover, each machine provides a coherent view of memory to all cores through a cache-coherence protocol [31]. The coherence protocol works at the granularity of a cache line (64 bytes), as each core tracks the status of a memory location and communicates to other cores when the location is modified or accessed. Unfortunately, maintaining cache-coherence is costly in NUMA machines because of the expensive communication through the interconnect. This latency cost is up to 2-3$\times$ higher than within a socket [32].

2.2 Ordering in Concurrency

With the advent of multicore machines, concurrent programming is the de facto approach to design today's algorithms, which applications rely upon. Hence, ordering becomes fundamental to the design of any concurrent algorithm. With ordering, algorithms can achieve varying levels of consistency to squeeze out the performance from hardware. For systems software, consistency depends on algorithms that require linearizability [27] for the composability of data structures [33, 6], concurrency control mechanisms for software transactional memory (`STM`) [1], or different isolation levels for database transactions [5, 34, 3]. The notion of ordering not only is limited to consistency, but also is applicable to either maintaining the history of operations for logging [35, 36, 37, 38] or determining the quiescence period for memory reclamation [39, 40, 33]. Depending on the consistency requirement, ordering such as a logical timestamping [33, 1], physical timestamping [39, 41, 6, 42, 43], and data-driven versioning [34, 44] can be achieved in several ways. The most common approach is to use a logical clock that is easier to maintain by software and is amenable to various ordering requirements.

A logical clock is a favorable ordering approach. An algorithm can globally update or access the clock in a single machine. However, the logical clock is one of the prime scalability bottlenecks on large multicore machines [7, 8] because it is maintained via an atomic instruction that incurs cache-coherence traffic. Unfortunately, these atomic instructions create cache-line contention that becomes the scalability bottleneck and it is more severe in multi-socket machines and in hyper-threaded scenarios [32, 45, 46]. Thus, maintaining a software clock is a deterrent to the scalability of an application, which holds true for several concurrency control mechanisms for concurrent programming [1, 47] and database transactions [3, 5]. To address this problem, various approaches—ranging from batched-update [5] to local versioning [34] to completely share-nothing designs [48, 49]—have been put into practice. However, these approaches have some side effects. They either increase the abort rates for `STM` [50] and databases [5] or complicate the design of the system software [49, 34, 51]. In addition, in the past decade, physical timestamping has been gaining traction for designing data structures [52, 43] synchronization mechanisms [6] for large multicore machines. However, such approaches assume that timestamp counters provided by hardware are synchronized, which is not entirely correct for existing hardware.

2.2.1 Multicore Hardware Clocks

Today's multicore machines provide per-core or per-processor hardware clocks [21, 22, 23, 24]. For example, all modern processor vendors such as `Intel` and `AMD` provide an `RDTSC` counter, while `Sparc` and `ARM` have `stick` and `cntvct` counter, respectively. These hardware clocks are **invariant** in nature, *i.e.*, processor vendors guarantee that these clocks are monotonically increasing at a constant rate,[1] regardless of the processor speed and its fluctuation. Moreover, they reset to zero whenever a processor receives a RESET signal (i.e., reboots). To provide such an invariant property, the current hardware always synchronizes these clocks from an external clock on the motherboard [53, 26, 25]. This is required

[1]These clocks may increase at a different frequency than that of a processor.

because vendors guarantee that the clocks do not fluctuate even if a processor goes to deep sleep states, which is always ensured by the motherboard clock. However, vendors do not guarantee the synchronization of clocks across a processor (socket) boundary in a multicore machine [21], not even within a processor because there is no guarantee that each processor can receive the broadcasted RESET signal at the same instant either inside a socket or across them. In addition, these hardware clocks do not provide the notion of real time.

These current invariant clocks are not reliable enough to design concurrent algorithms. To use them with confidence, we need to have a clock synchronization mechanism that provides constant physical skew between clocks, or we measure our offset to synchronize these clocks in software. Unfortunately, none of clock synchronization approaches work because hardware vendors cannot provide physical skew for every processor, and we cannot measure using existing clock synchronization protocols [54, 55, 56, 57], as software measurement induces overhead, in which the measured offset may be greater than the physical offset.

2.3 Locking Primitives

Locks are one of the easiest approach to designing concurrent application. Figure 2.2 illustrates the increasing use of locks in the Linux OS. Similar to the evolution of multicore [7, 8], not only the use of locks has dramatically increased for fine-grained locking [58], but also they have evolved into several types to minimize the cost of handling critical sections. For instance, the current OSes rely on several types of locks, ranging from non-blocking (*e.g.*, spinlocks, read-write locks) to blocking (*e.g.*, mutex, read-write semaphores). This trend is also applicable in other concurrent applications, such as databases, and other OSes. Another interesting trend is that these locks have been continuously evolving, as shown in Table 2.1, to improve the scalability of OS. For instance, a simple spinlock has evolved from a simple test-and-set (`TAS`) lock to a para-virtualized `qspinlock` (a variant of MCS lock and also supports virtualized environment). Hence, we try to understand the implication of these primitives and their interplay with the task scheduler on application performance. We first

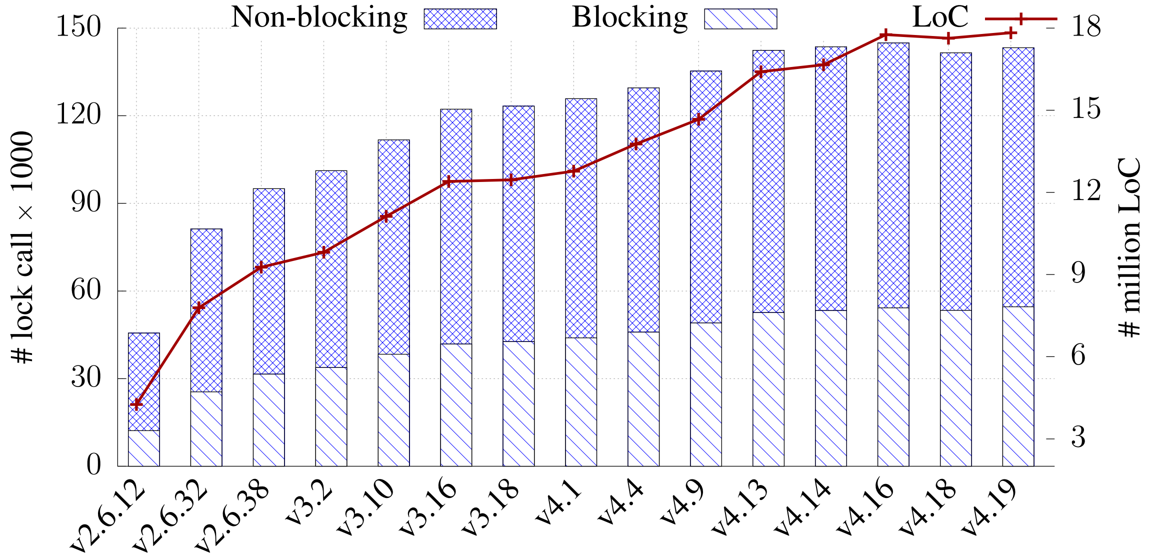

Figure 2.2: Indirect metric of the growing complexity of lock usage: the number of lock() API calls in the Linux kernel source code.

describe the general evolution of locks and later contrast it with lock evolution inside the Linux kernel.

2.3.1 Evolution of Lock design

Since the dawn of concurrent programming, hardware has been the dominant factor [32] in the evolution of lock algorithms. For instance, queue-based locks [59] reduce cache traffic relative to test-and-set (TAS) and ticket locks. On NUMA architectures, hierarchical locks improve throughput [60, 61, 62, 63, 64, 65] as they amortize remote access cost by physically partitioning a lock into a global lock and per-node locks. Cohort locks [63] generalize this design for any lock combination up to two levels, while Chabbi *et al*. [64, 65] further extended this approach to multiple hierarchies, using the MCS lock as a building block. Unfortunately, hierarchical locks have two issues: degraded performance for small numbers of cores, and, most importantly, memory overhead. AHMCS [65] addresses the problem of single thread performance but suffers from the memory overhead of hierarchical locks. Meanwhile, our CST locks work [66] partially addresses the problem of memory overhead by allocating the lock's memory on a first touch basis. However, it still suffers from

Table 2.1: Evolution of synchronization primitives in the last 15 years in Linux along with their introduction with the corresponding Linux version.

Locks	Features/Optimization	Reason:	Linux ver
spinlock (NB)	TAS	Simple test-and-set based lock that continously spins	v2.6.12
	Ticket	FIFO locks for fairness up to 255 cores	v2.6.25
	Ticket with para-virtualized TAS	Ticket lock in normal case, but TAS lock in para-virtualized scenario	v2.6.27
	Paravirtualized Ticket	Use Ticket locks in both para-virtualized and non-paravirtualized scenario	v2.6.30
	Paravirtualized Ticket with slowpath	Schedule out vCPUs in paravirtualized case to mitigate LHP/LWP	v3.11
	qspinlock with paravirtualized TAS	4 byte MCS lock to remove cache-line contention, but TAS to remove LHP/LWP	v4.4
	Unfair paravirtualized qspinlock	Limited lock stealing and adaptive lock spinning to mitigate LHP/LWP	v4.5
rwlock (NB)	4 byte TAS	Simple test-and-set lock with first byte for storing reader count	v2.6.12
	4 byte TAS with 4 byte read count	Increase the readers count to compensate for larger machines	v3.11
	Ticket with 4 byte read/write (qrwlock)	Improves the throughput and latency of the rwlock	v3.16
	qspinlock with four byte read-/write counter (qrwlock)	Further removes cache-line contention of the Ticket lock	v4.4
seqlock (NB)	spinlock with sequence number	Non-starving writers with retrying readers to read consistent data	v2.6.12
	New locking reader type method	Allow readers to acquire the lock without updating the sequence count	v3.12
	New latch API	Allows queries during non-atomic modifications with two versions	v4.2
mutex (B)	A counter with a spinlock and a wait list	A semaphore with a wait queue that maintains waiters	v2.6.12
	atomic counter with a waiting queue	An atomic counter with a simple wait queue to handle waiters	v2.6.16
	Optimistic spinning	Allowed waiters to spin for a time period before scheduling out	v2.6.30
	Queue-based optimistic spinning	Allow waiters to form a queue while spinning to reduce cache-line contention	v3.16
	Lockless next waiter wakeup	Shortens the fast path to wake-up the very next waiter, thereby decreasing spinlock contention	v4.5
	Break if vCPU is preempted	Schedule out in the mid-path phase if the owner vCPU is preempted	v4.9
rwsem (B)	A counter with a spinlock and a wait list	A counter maintains both writers and readers with wait list for waiters	v2.6.12
	Optimistic spinning	Allow waiters to wait while spinning to mitigate blocked-waiter wakeup problem	v3.15
	Writer optimistic spinning	Disallow waiting writers to waste CPU cycles	v4.7
percpu-rwsem (B)	percpu counter with mutex	Lightweight RCU-based readers, with rcu_synchronize()-based mutex writers	v3.6
	rwsem with percpu and global counter	Optimize readers' counter latency after a writer unlocks	v3.7
	Optimized readers	Removed the global counter by ordering reader-state vs reader-count with RCU	v4.8

the memory overhead in the worst case and introduces a large memory allocate overhead on the critical path. Unfortunately, both of these works partially address one of these issues, but not both concurrently.

We observe a similar evolution in designing readers-writer locks. Mellor-Crummey and Scott [67] proposed variants of readers-writer locks on top of the queue-based locks. However, these locks create coherence traffic in NUMA machines. Calciu *et al.* [68] proposed a per-socket read indicator on top of Cohort locks to localize the reader contention within a socket, but both per-socket or per-CPU [69] approaches require extra memory and are beneficial only in particular cases [70, 71, 72].

2.3.2 Locks in the kernel space (Linux)

Over the past decade, the Linux kernel has been striving for more concurrency by switching to finer granularity locks, as shown in Table 2.1. One of the most significant goals is to maintain optimal single-thread performance. In addition, lock design must consider: 1) the interaction with the scheduler, 2) the size of the lock structure, and 3) avoiding any explicit memory allocation. These factors have led to sophisticated optimizations. The `spinlock` is the primary locking construct in Linux; it has evolved from `TAS` to ticket locks to an MCS variant [73]. The current design is an amalgamation of two locks: a `TAS` lock in the fast path and an MCS lock in the slow path.

The second most widely used synchronization primitives are blocking locks. Moreover, various OSes, in particular Linux, do not allow nested critical sections for any blocking locks. Currently, these locks can be considered as the big-kernel locks in Linux, as they are the most common guards that an OS uses for guarding inodes in file system, and address space manipulation. There are two variants: `mutex` and `rwsem`.

`mutex` incorporates a fast path comprising of test-and-test-and-set lock (TTAS), an abortable queue-based spinning in mid-path [74], and a parking list per-lock instance in the slow path. The algorithm works by first trying to atomically update the lock variable, called fast path; on failure, the mid-path phase (optimistic spinning) begins in which only a single waiter is queued up if there is no spinning waiter and optimistically spins until its schedule quota expires. If the waiter still does not acquire the lock, it goes to the slow-path phase in which it acquires a lock on the parking list (parking lock), adds itself, and schedules out after releasing the parking lock. During the unlock phase, the lock holder first resets the TTAS variable and wakes up a waiter from the parking list while holding the parking lock. Meanwhile, it is possible that either a new waiter can acquire the lock in the fast path or a spinning waiter in the mid path. Now, once a waiter is scheduled in, it again acquires the parking lock and tries to acquire the TTAS lock. If successful, it removes itself from the parking list and enters the critical section; otherwise, it schedules itself out again and

sleeps until a lock holder wakes it up. The current algorithm is unfair because of the TTAS lock; even starves its waiters in the slow-path phase. Moreover, the algorithm also suffers two overheads. Firs is from cache-line contention because of the TTAS lock and waiters maintenance. Second is the scheduling overhead in the slow-path phase and the unlock phase for parking a running waiter and waking up a sleeping waiter [66].

The readers-writer semaphore (`rwsem`) is an extension of `mutex`, with a writer-preferred version [75, 76]. Both the write lock and the readers count are encoded in a word to decide readers, writer, and waiting writers or readers. Moreover, `rwsem` maintains a single parking list in which both readers and writers are added in the slow path. Similar to `mutex`, it also suffers from severe cache-line movement both when the cores are over- and under-subscribed because of the contention on the reader-writer indicator.

In summary, these both blocking and non-blocking locks are the most widely used locking primitives. Unfortunately, these locks degrade the performance of applications. We particularly study the scalability bottlenecks in OS by focusing on one component: the file system. We design a new benchmarking suite, called `FxMark`, that is a collection of micro- and macro-benchmarks that stresses various components of file systems. We now discuss the identified locking bottlenecks in five widely deployed file systems.

2.3.3 Locking Bottlenecks in Deployed File Systems

Table 2.2 presents the lock-based scalability bottlenecks with our tool (`FxMark`), which found 22 bottlenecks in five widely deployed file systems. We draw the following observations, to which I/O-intensive applications must pay close attention:

- `rename()` is commonly used in many applications for transactional updates [79, 80, 81]. However, we found that `rename()` operations are completely serialized at a system level by two locks in Linux for consistent updates of the `dentry` cache.
- All of the tested file systems hold an exclusive lock for a file (`inode->i_mutex`) during a write operation. This is a critical bottleneck for high-performance database

Table 2.2: Identified lock-based scalability bottlenecks in tested file systems with FxMark [77].

FS	Bottleneck	Sync. object	Scope	Operation
VFS	Rename lock	rename_lock	System	rename()
	inode list lock	inode_sb_list_lock	System	File creation and deletion
	Directory access lock	inode->i_mutex	Directory	All directory operations
	dentry lockref [78]	dentry->d_lockref	dentry	Path name resolution
btrfs	Acquiring a write lock for a B-tree node	btrfs_tree_lock()	File system	All write operations
	Acquiring a read lock for a B-tree node	btrfs_set_lock_blocking_rw()	File system	All read operations
	Checking data free space	data_info->lock	File system	File append
	Reserving data blocks	delalloc_block_rsv->lock	File system	File append
	File write lock	inode->i_mutex	File	File write
ext4	Acquiring a read lock for a journal	journal->j_state_lock	Journal	Heavy metadata update operations
	Orphan inode list	sbi->s_orphan	File system	File deletion
	Block group lock	bgl->locks	Block group	File creation and deletion
	File write lock	inode->i_mutex	File	File write
F2FS	Single-threaded writing	sbi->cp_rwsem	File system	File write
	SIT (segment information table)	sit_i->sentry_lock	File system	File write
	NAT (node address table)	nm_i->nat_tree_lock	File system	File creation, deletion, and write
	File write lock	inode->i_mutex	File	File write
tmpfs	Capacity limit check (per-CPU counter)	sbinfo->used_blocks	File system	File write near disk full
	File write lock	inode->i_mutex	File	File write
XFS	Journal writing	log->l_icloglock	File system	Heavy metadata update operations
	Acquiring a read lock of XFS inode	ip->i_iolock	File	File read
	File write lock	inode->i_mutex	File	File write

systems allocating a large file but not maintaining the page cache by themselves (e.g., PostgreSQL). Even in the case of using direct I/O operations, this is the critical bottleneck for both read and write operations.

- Operations such as creating, deleting, growing, and shrinking a file are not scalable in Linux. The key reason is that existing consistency mechanisms (i.e., journaling in ext4 and XFS, copy-on-write in btrfs, and log-structured writing in F2FS) are not designed with multicore scalability in mind.
- Some file systems have peculiar scalability characteristics for some operations; a single file read of multiple threads is not scalable in XFS due to the scalability limitation of Linux's read/write semaphore; in F2FS, a checkpointing operation caused by segment cleaning freezes entire file system operations so scalability will be seriously hampered under write-heavy workloads with low free space; enumerating directories in btrfs is not scalable because of too frequent atomic operations. The scalability of an I/O-intensive application is very file system-specific.
- In file systems, concurrent operations are coordinated mainly by locks. Because

of spinning of spinlock and optimistic spinning of blocking locks, non-scalable file systems tend to consume more CPU cycles to alleviate the contention. In our benchmarks, about 30-90% of CPU cycles are consumed for synchronization.

In summary, `FxMark` points out that some of the file system operations are inherently non-scalable. However, some of the operations suffer from inefficient locking primitives, as shown in Table 2.2. Thus, we need scalable locking primitives, both blocking and non-blocking, that do not degrade the performance of applications.

We now discuss how task scheduling impacts the performance of applications that use various synchronization primitives to guard shared resources. In particular we focus on the problem of double scheduling that occurs when multiple task schedulers are stacked together. This thesis considers the case of virtualized environment to discuss various issues with the task scheduling problem.

2.4 Double Scheduling in VMs

Double scheduling is a phenomenon in which two schedulers are stacked on top. In the case of virtualized environment, the VM schedules processes on `vCPUs` and the hypervisor schedules `vCPUs` on physical CPUs. One of the major issues with double scheduling or any n-level scheduling is there is a missing semantic information among task schedulers. The semantic gap introduces forward progress of applications because a `vCPU` can be evicted, while executing a critical task. We describe several symptoms in the form of preemption problems that occur in a virtualized environment.

Symptoms of Double Scheduling. In a virtualized environment, a hypervisor multiplexes the hardware resources for a VM, such as assigning `vCPUs` to physical CPUs (`pCPUs`). In particular, it runs a `vCPU` to execute by its fair share [82], which is a general policy of commodity OSes such as Linux, and preempts it because of either `vCPUs` of other VM or of the intermittent processes of the OS and bookkeeping tasks of the hypervisor such as I/O threads. Hence, there is a possibility that the hypervisor can preempt a `vCPU` while executing

some critical task inside a VM that leads to an application performance anomaly, which we enumerate below:

❶ **Lock holder preemption (LHP)** problem occurs when a vCPU holding a lock gets preempted and all waiters waste CPU cycles for the lock. Most of the prior works [83, 84, 85, 86] have focused on non-blocking primitives such as spinlocks.[2] On the other hand, LHP also occurs in blocking primitives such as mutex [89] and rwsem [90, 71], which the prior works have not identified. However, LHP accounts up to 90% preemptions for blocking primitives in some of the memory intensive applications that have short critical sections.

❷ **Lock waiter preemption (LWP)** problem stems when the very next waiter is preempted just before acquiring the lock, which occurs due to the strict FIFO ordering of spinlocks [85, 86]. Fortunately, this problem has been mostly mitigated in existing spinlock design [87, 91], as the current implementation allows waiters to steal the lock before joining the waiter queue. We do not see such a problem in blocking primitives because the current implementation is based on the TTAS lock—an unfair lock, which inherently mitigates LWP.

❸ **Blocked-waiter wakeup (BWW)** problem occurs mostly for blocking primitives in which the latency to wake up a waiter to pass the lock is quite high. This issue severely degrades the throughput of applications running on a high core count [66], even in a native environment. Moreover, it is evident in both under- and over-committed VM scenarios. For example, the BWW problem degrades the application scalability up to 60%.

❹ **Readers preemption (RP)** problem is a new class of problem that occurs when a vCPU holding a read lock among multiple readers gets preempted. This problem impedes the forward progress of a VM and also increases the latency of the write lock. For instance, various memory-intensive workloads have sub-optimal throughput as RP accounts to at most 20% of preemptions. We observe this issue in various read-dominated memory-intensive workloads in which the readers are scheduled out.

[2] Non-blocking locks, both holders and waiters, do not schedule out. However, the para-virtualized interface converts spinlocks to blocking locks (only waiters) with hypercalls [87, 88] to overcome LHP/LWP issues.

❺ **RCU reader preemption (RRP)** problem is a type of RP problem that occurs when an RCU reader is preempted, while holding the RCU read lock [92]. Because of RRP, the guest OS suffers from an increased quiescence period. This issue can increase the memory footprint of the application, and is responsible for 5% of preemptions.

❻ **Interrupt context preemption (ICP)** problem happens when a vCPU that is executing an interrupt context gets preempted. In particular, this problem is different from prior works that focus on interrupt delivery [93, 94] rather than interrupt handling. This issue occurs in cases such as TLB shootdowns, function call interrupts, rescheduling interrupts, IRQ work interrupts, etc. in every commodity OS. For example, we found that Apache web server, an interrupt-intensive workload, suffers from the ICP problem as it accounts to almost 18% of preemptions for evaluated workloads.

2.5 Conclusion

In summary, this chapter showed how scheduling of events: from ordering in concurrency algorithms, waiters in lock algorithms, to stacked task scheduling impacts the performance of applications. At a high level, applications are suffering from synchronization primitives that degrade the performance of applications on large multicore machines. This happens because these primitives rely on primitives that suffer from higher overhead and do not cater to the evolving hardware and software requirements. Moreover, levels of indirection introduce semantic gap, which affects the performance of these primitives, thereby affecting the performance of applications.

CHAPTER 3

ORDERING PRIMITIVE

This chapter presents a scalable timestamping primitive, called `Ordo`, by employing **invariant hardware clocks**, which current major processor architectures already support [21, 22, 23, 24]. An invariant clock has a unique property: It is monotonically increasing and has a constant skew, regardless of dynamic frequency and voltage scaling, and resets itself to zero whenever a machine is reset (on receiving the RESET signal), which is ensured by the processor vendors [25, 26, 22, 24]. However, assuming that all invariant clocks in a machine are synchronized is incorrect because processors do not receive the RESET signal at the same time, which makes these clocks unusable. Thus, we cannot compare two clocks with confidence when correctly designing any time-based concurrent algorithms. Moreover, we cannot exactly synchronize these clocks because 1) the conventional clock algorithms [56, 57, 54] provide only a time bound that can have a lot of overhead, and 2) the hardware vendors do not divulge any communication cost that is required to establish strict time bounds for software clock synchronization to work effectively.

The comparison of clocks with no confidence makes them unusable to correctly design a concurrent algorithm. Thus, to provide a guarantee of correct use of these clocks, we propose a new primitive, called `Ordo`, that embraces uncertainty when we compare two clocks and provides an illusion of a *globally synchronized hardware clock* in a single machine. However, the notion of a globally synchronized hardware clock is only feasible if we can measure the offset between clocks. Unfortunately, accurately measuring this offset is difficult because hardware does not provide minimum bounds for the message delivery [56, 57, 54]. To solve this problem, we exploit the unique property of invariant clocks and empirically define the uncertainty window by utilizing one-way-delay latency among clocks.

Under the assumption of invariant clocks, `Ordo` provides an uncertainty window that

```
1 def get_time(): # Get timestamp without memory reordering
2   return hardware_timestamp()  # Timestamp instruction
3
4 def cmp_time(time_t t1, time_t t2): # Compare two timestamps
5   if t1 > t2 + ORDO_BOUNDARY:   # t1 > t2
6     return 1
7   elif t1 + ORDO_BOUNDARY < t2: # t1 < t2
8     return -1
9   return 0                      # Uncertain
10
11 def new_time(time_t t): # New timestamp after ORDO_BOUNDARY
12   while cmp_time(new_t = get_time(), t) is not 1:
13     continue # pause for a while and retry
14   return new_t # new_t is greater than (t + ORDO_BOUNDARY)
```

Listing 3.1: `Ordo` clock API. The `get_time()` method returns the current timestamp without reordering instructions.

remains constant while a machine is running. Thus, `Ordo` enables algorithms to become multicore friendly by either replacing the software clock or correctly using a timestamp with a minimal core-local computation for an ordering guarantee. We find that various timestamp-based algorithms can be simplified as well as benefit from our `Ordo` primitive, which has led us to design a simple API. The only trick lies in handling the uncertainty window, which we explain for both physical and logical timestamp-based algorithms such as a concurrent data structure library, and concurrency control algorithms for `STM` and databases. With our `Ordo` primitive, we improve the scalability of various algorithms and systems software (e.g., `RLU`, `OCC`, `Hekaton`, `TL2`, and process forking) up to 39.7× across four architectures: `Intel Xeon` and `Xeon Phi`, `AMD`, and `ARM`, while maintaining equivalent performance in optimized scenarios. Moreover, our version of the conventional `OCC` algorithm outperforms the state-of-the-art algorithm by 24% in a lightly contended case for the TPC-C workload.

3.1 `Ordo`: A Scalable Ordering Primitive

`Ordo` relies on invariant hardware clocks to provide an illusion of a globally synchronized hardware clock with some uncertainty. To provide such an illusion, `Ordo` exposes a simple API that allows applications either to obtain a global timestamp or to order events with some uncertainty. Thus, we introduce an approach to measure the uncertainty window, followed

by a proof to ensure its correctness.

3.1.1 Embracing Uncertainty in Clock: `Ordo` API

Timestamp-based concurrent algorithms can reliably use an invariant clock if we define the uncertainty period. Moreover, such algorithms are designed to execute on two or more cores/threads, which require two important properties from invariant clocks: 1) establishing a relation among two or more cores to compare events and 2) providing a notion of a monotonically increasing globally synchronized clock to order events in a single machine. Thus, we propose `Ordo`, which provides a notion of a monotonically increasing timestamp but also exposes an uncertainty window, called `ORDO_BOUNDARY`, in which we are unsure of the ordering. To ease the use of our `Ordo` primitive, we expose three simple methods (Listing 3.1) for such algorithms:

- **`get_time()`** is the hardware-specific timestamping instruction that also takes care of the hardware-specific requirements such as instruction reordering.
- **`new_time(time_t t)`** returns a new timestamp at the granularity of `ORDO_BOUNDARY` and is greater than `t`.
- **`cmp_time(time_t t1, time_t t2)`** establishes the precedence relation between timestamps with the help of `ORDO_BOUNDARY`. If the difference between `t1` and `t2` is within `ORDO_BOUNDARY`, it returns 0, meaning that we are uncertain.

3.1.2 Measuring Uncertainty between Clocks: Calculating `ORDO_BOUNDARY`

Under the assumption of invariant clocks, the uncertainty window (or skew) between clocks is constant because both clocks monotonically increase at the same rate but may receive a RESET signal at different times. We define this uncertainty window as a physical offset: Δ. A common approach to measure Δ is to use a clock-synchronization mechanism, in which a clock reads its value and the value of other clocks, computes an offset, and then adjusts its clock by the measured offset [95]. However, this measurement introduces various errors,

```
1  runs = 100000  # multiple runs to minimize overheads
2  shared_cacheline = {"clock": 0, "phase": INIT}
3
4  def remote_worker():
5    for i in range(runs):
6      while shared_cacheline["phase"] != READY:
7        read_fence() # flush load buffer
8      ATOMIC_WRITE(shared_cacheline["clock"], get_time())
9      barrier_wait() # synchronize with the local_worker
10
11 def local_worker():
12   min_offset = INFINITY
13   for i in range(runs):
14     shared_cacheline["clock"] = 0
15     shared_cacheline["phase"] = READY
16     while shared_cacheline["clock"] == 0:
17       read_fence() # flush load buffer
18     min_offset = min(min_offset, get_time() -
19                      shared_cacheline["clock"])
20     barrier_wait() # synchronize and restart the process
21   return min_offset
22
23 def clock_offset(c0, c1):
24   run_on_core(remote_worker, c1)
25   return run_on_core(local_worker, c0)
26
27 def get_ordo_boundary(num_cpus):
28   global_offset = 0
29   for c0, c1 in combinations([0 ... num_cpus], 2):
30     global_offset = max(global_offset,
31                         max(clock_offset(c0, c1),
32                             clock_offset(c1, c0)))
33   return global_offset
```

Figure 3.1: Algorithm to calculate the `ORDO_BOUNDARY`: a system-wide global offset.

such as reading remote clocks and software overheads, including network jitter. Thus, we cannot rely on this method to define `ORDO_BOUNDARY` because 1) hardware vendors do not provide minimum bounds on the message delivery; 2) the algorithm is erroneous because of the uncertainty of the message delivery [56, 57, 54];[1] and 3) periodically using the clock synchronization will waste CPU cycles, which will increase with core count. Moreover, the measured offset can be either larger or smaller than Δ, which renders it unusable for concurrent algorithms.

[1] Although clock synchronization algorithms are widely used in distributed systems settings where multiple clocks are in use, they are also applicable in a single machine with per-core clocks, and it maintains the notion of a global clock by synchronizing itself with an outside clock over the network [56].

Measuring global offset

Instead of synchronizing these clocks among themselves, we exploit their unique *invariant* property and empirically calculate a system-wide global offset, called `ORDO_BOUNDARY`, which ensures that the measured offset between clocks is always greater than their physical offset. We define *measured offset* (δ_{ij}) as a one-way-delay latency between clocks (c_i to c_j), and this measured offset (δ) will be greater than the physical offset because of the extra one-way-delay latency.

Figure 3.1 illustrates our algorithm to calculate the global offset after calculating the correct pairwise offset for all clocks in a machine. We measure the offset between core c_i and c_j as follows: c_i atomically writes its timestamp value to the variable (line 8), which notifies waiting c_j (line 16). On receiving the notification, c_j reads its own timestamp (line 18) and then calculates the difference (line 19), called δ_{ij}, and returns the value. The measured offset has an extra error over the physical offset because δ_{ij}, between cores c_i and c_j, also includes software overhead, interrupts, and the coherence traffic. We reduce this error by calculating the offset multiple times (line 1) and taking the minimum of all runs (line 19–21). To define the correct offset between c_i and c_j, we calculate offset from both ends (i.e., δ_{ij} and δ_{ji}) and choose their maximum since we do not know which clock is ahead of the other (line 31). After calculating pairwise offsets among all cores, we select the maximum offset as the global offset, or `ORDO_BOUNDARY`, (line 30). We take the maximum offset among all cores because it ensures that any arbitrary core is guaranteed to see a new timestamp once the `ORDO_BOUNDARY` window is over, which enables us to compare timestamps with confidence. Moreover, the calculated `ORDO_BOUNDARY` is reasonably small because we use cache coherence as our message delivery medium, which is the fastest means of communication between cores.

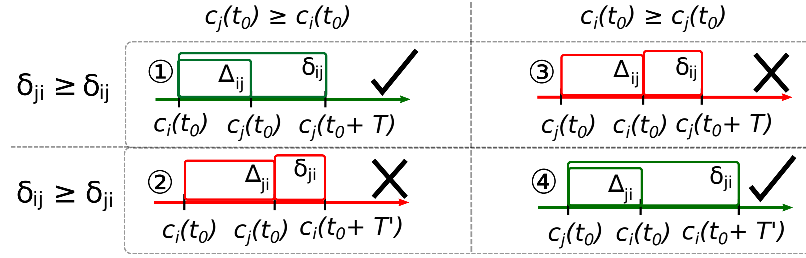

Figure 3.2: Calculating offset ($\delta_{i\leftrightarrow j}$) using pairwise one-way-delay latency between clocks ($\mathtt{c}_i$ and $\mathtt{c}_j$). Δ_{ij} and Δ_{ji} are the physical offsets between $\mathtt{c}_i$ and $\mathtt{c}_j$, and $\mathtt{c}_j$ and $\mathtt{c}_i$, respectively. δ_{ij} and δ_{ji} are measured offsets with our approach. Depending on physical and measured offsets, there are four possible cases. Unlike existing clock-synchronization protocols [56, 57, 54, 95][3] that average the calculated latency based on RTT, we consider each direction separately to measure the offset and consider only the direction that is always greater the positive physical offset, such as cases 1 and 4.

Correctness of the measured offset

For invariant clocks to be useful, our approach to calculate the `ORDO_BOUNDARY` must have the following invariant:

Invariant:

The global measured offset is always greater than the maximum physical offset between cores.

We first state our assumptions and prove with a lemma and a theorem that our algorithm (Figure 3.1) to calculate the global offset is correct with respect to invariant clocks.

Assumptions. Our primary assumption is that clocks are invariant and have a physical offset that remains constant until the machine is reset. Moreover, the hardware's external clock-synchronization protocol maintains the monotonic increase of these clocks because it guarantees an invariant clock [25, 26, 53], which is a reasonable assumption for modern, mature many-core processors such as X86, `Sparc`, and `ARM`. Hence, with the invariant timestamp property, the physical offset, or skew, between clocks also remains constant.

Lemma

The maximum of calculated offsets from $\mathtt{c}_i$ to $\mathtt{c}_j$ and $\mathtt{c}_j$ to $\mathtt{c}_i$ pairs (i.e., $\delta_{i\leftrightarrow j} = \max(\delta_{ij}, \delta_{ji})$) is always greater than or equal to the physical offset (i.e., Δ_{ij}).

Proof

For cores c_i and c_j:

$$\delta_{ij} = |c_j(t_1) - c_i(t_0)| \qquad = |c_j(t_0 + T) - c_i(t_0)| \tag{3.1}$$

$$\delta_{ji} = |c_i(t_1) - c_j(t_0)| \qquad = |c_i(t_0 + T') - c_j(t_0)| \tag{3.2}$$

δ_{ij} and δ_{ji} denote the offset measured from c_i to c_j and c_j to c_i, respectively, and $|\Delta_{ij}| = |\Delta_{ji}|$. $c_i(t_0)$ and $c_j(t_0)$ denote the clock value at a real time t_0, which is a constant monotonically increasing function with a defined timestamp frequency by the hardware. T and T' are the recorded true time when they receive a message from another core (Figure 3.1: line 16), which denotes the one-way-delay latency in either direction (Figure 3.1: line 31). Depending on the relation between clocks at time t ($c_i(t)$ and $c_j(t)$) and measured offsets (δ_{ij} and δ_{ji}) from Equation 3.1 and 3.2, Figure 3.2 presents four possible cases: ① and ② if $c_j(t) \geq c_i(t)$, and ③ and ④ if $c_i(t) \geq c_j(t)$. Let us consider a scenario in which $c_j(t) \geq c_i(t)$ representing cases ① and ② and assume that we obtain case ②. By contradiction, case ② is not possible because our algorithm adds an extra cost of cache-line transfer (line 16) to measure the offset of constant monotonically increasing clocks, which will always return case ①; thus, δ_{ji} is going to be greater than δ_{ij}. Since we do not know which clock is lagging, we calculate other possible scenarios, ($c_i(t) \geq c_j(t)$: cases ③ and ④) so that we always obtain the maximum of two measured offsets without any physical offset information (lines 31–32). Thus, either case ① or case ④ always guarantees that measured offset is greater than the physical offset. □

Theorem

The global offset is the maximum of all the pairwise measured offsets for each core.

Proof

$\delta^g = \max(\delta_{i \leftrightarrow j}) = \max(\max(\delta_{ij}, \delta_{ji})) \mid \forall i, j \in \{0..N\}$.

N is the number of cores and δ^g is the `ORDO_BOUNDARY`. We extend the lemma to all pairwise combinations of cores (refer to Figure 3.1 line 29) to obtain the maximum offset value among all cores. This approach 1) establishes a relation among all cores such that if any core wants to have a *new timestamp*, it can obtain such a value at the expiration of `ORDO_BOUNDARY` and 2) also establishes an uncertainty period at a global level in which we can definitely provide the precedence relationship among two or more timestamps if all of them have a difference of the global offset; otherwise, we mark their comparison as uncertain if they fall in that offset window. □

Logical Timestamping: CC Algorithm

```
txn_read/write(txn):
    txn.timestamp = get_timestamp()
    # Add to read/write set
txn_commit(txn):
    lock_writeset(txn)
    commit_txn = get_new_timstamp()
    if validate_readset(txn, commit_txn) is True:
        commit_write_changes(txn, commit_txn)
```

Physical Timestamping: oplog

```
op_log(op):
    local_log = get_local_log()
    thd.log.append([op, get_timestamp()])
op_synchronize(op):
    # Acquire per-object lock
    current_log = []
    for log in all_logs():
        # Add all op object to the current_log
    current_log.sort() # Sort via timestamp
    apply(current_log) # Apply all op objects
```

Listing 3.2: Pseudo-code of logical timestamping algorithms used in `STM` [1, 2] and databases [3, 4, 5], and that of physical timestamping used in `Oplog` [6].

3.2 Algorithms with `Ordo` without Uncertainty

With the `Ordo` API, an important question is *how do we handle the uncertainty exposed by the `Ordo` primitive while adopting it for timestamp-based algorithms*? We answer this question by classifying them into two categories:

- **Physical to logical timestamping:** Algorithms that rely on logical timestamping or versioning (refer to Listing 3.2) will require all three methods, but the most important one is the `cmp_time()`, which provides the ordering between clocks to ensure their correctness. By introducing the `ORDO_BOUNDARY`, we now need to extend these algorithms to either defer or wait to execute any further operations.
- **Hardware timestamping:** Algorithms that use physical timestamping [43, 6] can use `new_time()` method to access a globally synchronized clock to ensure their correctness under invariant clocks.

3.2.1 Read-Log-Update (`RLU`)

`RLU` is an extension to the RCU framework that enables a semi-automated method of concurrent read-only traversals with multiple updates. It is inspired by STM [96] and maintains an object-level write-log per thread, in which writers first lock the object and then copy those objects to their own write log and manipulate them locally without affecting the original structure. `RLU` adopts the RCU barrier mechanism with a global clock-based logging mechanism [1, 50].

Listing 3.3 illustrates the pseudo-code of the functions of `RLU` that use the global clock. We refer to the pseudo-code to explain its workings, limitations, and our changes to the `RLU` design. `RLU` works as follows: All operations reference the global clock in the beginning (line 2) and rely on it to dereference the shared objects (line 15). For a write operation, each writer first dereferences the object and copies it in its own log after locking the object. At the time of commit (line 25), it increments the global clock (line 27), which effectively splits

```
1    # All operations acquire the lock
2    def rlu_reader_lock(ctx):
3      ctx.is_writer = False
4      ctx.run_count = ctx.run_count + 1 # Set active
5      memory_fence
6  -   ctx.local_clock = global_clock # Record global clock
7  +   ctx.local_clock = get_time() # Record Ordo global clock
8
9    def rlu_reader_unlock(ctx):
10     ctx.run_count = ctx.run_count + 1 # Set inactive
11     if ctx.is_writer is True:
12       rlu_commit_write_log(ctx) # Write updates
13 -------------------------------------------------------------
14   # Pointer dereference
15   def rlu_dereference(ctx, obj):
16     ptr_copy = get_copy(obj) # Get pointer copy
17     # Return object or object copy
18     other_ctx = get_ctx(thread_id) # check for stealing
19 -   if other_ctx.write_clock <= ctx.local_clock:
20 +   if cmp_time(ctx.local_clock, other_ctx.write_clock) == 1:
21       return ptr_copy # return ptr_copy (stolen)
22     return obj # no stealing, return the oject
23 -------------------------------------------------------------
24   # Memory commit
25   def rlu_commit_write_log(ctx):
26 -   ctx.write_clock = global_clock + 1 # Enable stealing
27 -   fetch_and_add(global_clock, 1) # Advance clock
28 +   # Ordo clock with an extra ORDO_BOUNDARY for correctness
29 +   ctx.write_clock = new_time(ctx.local_clock + ORDO_BOUNDARY)
30     rlu_synchronize(ctx) # Drain readers
31     rlu_writeback_write_log(ctx) # Safe to write back
32     rlu_unlock_write_log(ctx)
33     ctx.write_clock = INFINITY # Disable stealing
34     rlu_swap_write_logs(ctx) # Quiesce write logs
35 -------------------------------------------------------------
36   # Synchronize with all threads
37   def rlu_synchronize(ctx):
38     for thread_id in active_threads:
39       other = get_ctx(thread_id)
40       ctx.sync_cnts[thread_id] = other.run_cnt
41     for thread_id in active_threads:
42       while:
43         if ctx.sync_cnts[thread_id] & 0x1 != 0:
44           break # not active
45         other = get_ctx(thread_id)
46         if ctx.sync_cnts[thread_id] != other.run_cnt:
47           break # already progressed
48 -       if ctx.writer_clock <= other.local_clock:
49 +       if cmp_time(other.local_clock, ctx.writer_clock) == 1:
50           break # started afer me
```

Listing 3.3: RLU pseudo-code including our changes.

the memory snapshot into the old and new one. While old readers refer to the old snapshot that have smaller clock values than the incremented global clock, the new readers read the new snapshot that starts after the increment of the global clock (lines 18 – 22). Later, after increasing the global clock, writers wait for old readers to finish by executing the RCU-style quiescence loop (lines 41 – 50), while new operations obtain new objects from the writer log. As soon as the quiescence period is over, the writer safely writes back the new objects from its own log to the shared memory (line 31) and then releases the lock (line 32). In summary, `RLU` has three scalability bottlenecks: 1) maintain and reference the global clock, which does not scale with increasing core count; 2) lock/unlock operation on an object, and 3) copy an object for a write operation. `RLU` tries to mitigate the first problem by employing a defer-based approach, but it comes at the cost of extra memory utilization. The last two are design choices that prefer programmability over the hand-crafted copy management mechanism.

We address the scalability bottleneck of the global clock with the `Ordo` primitive. Now, every read and write operation refers to the global clock via `get_time()` (line 6), and relies on it to dereference the object by comparing the timestamp for two contexts with the `cmp_time()` method (line 7), which provides the same comparison semantics before the modification (line 6). At the time of commit, we demarcate between the old and new snapshot by obtaining a new timestamp via the `new_time()` method (line 29), and later rely on this timestamp to maintain the clock-based quiescence period for readers that are still using the old snapshot (line 49). Note that we add an extra `ORDO_BOUNDARY` to correctly differentiate between the old snapshot and the newer one, as we may obtain an incorrect snapshot if one of the clocks has negative skew.

Our modification does not break the correctness of the `RLU` algorithm, i.e., to always have a consistent memory snapshot in the `RLU` protected section. In other words, at time $t' > t$; because of the constant monotonically increasing clock, the time obtained at the commit time of the writer (line 29) is always greater than the previous write timestamp,

thereby keeping a protected `RLU` section from seeing its concurrent overwrite. There can be an inconsistency if a thread has just updated a value and another thread is trying to steal the object while having a negative skew than the committed thread's clock. In this case, the reading thread may read an old snapshot, which can have an inconsistent memory snapshot. Since `RLU` only supports only a single version, we address this issue by adding an extra `ORDO_BOUNDARY` at the commit time, which ensures that we have at least one `ORDO_BOUNDARY` difference between or among threads. Moreover, our modification enforces the invariant for writers in the following ways: 1) If a new writer is able to steal the copy from the other writer (line 20), it still holds the invariant; 2) If a new writer is unable to steal the copy of the locked object (line 22), the old writer will quiesce while holding the writer lock (line 37), which will force the new writer to abort and retry because `RLU` does not allow writer-writer conflict. Note that the `RLU` algorithm already takes care of the writer log quiescence by maintaining two versions of logs, which are swapped to allow stealing readers to use them (line 34).

3.2.2 Concurrency Control for Databases

Database systems serve highly concurrent transactions and queries, with strong consistency guarantees by using concurrency control (CC) schemes. Two main CC schemes are popular among state-of-the-art database systems: optimistic concurrency control (`OCC`) and multi-version concurrency control (`MVCC`). Both schemes use timestamps to decide global commit order without locking overhead [97]. Listing 3.2 presents the pseudo-code of CC algorithms. **`OCC`** is a single CC scheme that consists of three phases: 1) *read*, 2) *validation*, and 3) *write*. In the first phase, a worker keeps footprints of a transaction in local memory. At commit time, it acquires locks on the write set. After that, it checks whether the transaction violates serializability in the *validation* phase by assigning a global commit timestamp to the transaction and validates both the read and write set by comparing their timestamps with the commit timestamp. After the *validation* phase, the worker enters the *write* phase in

which it makes its write set visible by overwriting original tuples and releasing held locks. To address the problem of logical timestamps in `OCC` [3], some state-of-the-art `OCC` schemes have mitigated the updates with either conservative read validation [5, 98] or data-driven timestamping [34].

To show the impact of `Ordo`, we modify the first two phases—read and validation phases—of the `OCC` algorithm [3]. In the *read phase*, we assign the timestamp via `new_time()`, which guarantees that the new timestamp will be greater than the previous one. The validation scheme is the same as before, but the difference is that `get_time()` provides the commit timestamp. The worker then uses the commit timestamp to validate the read set by comparing it with the recorded timestamp of both the read and write set. We apply a conservative approach of aborting the transactions if two timestamps fall within the `ORDO_BOUNDARY` in the validation step. Two requirements ensure serializability: 1) obtain a new timestamp that `new_time()` ensures, and 2) handle the uncertainty window, in which we conservatively abort transactions, thereby resolving the later-conflict check [99] to ensure the global ordering of transactions.

`MVCC` is another major category of CC schemes. Unlike other single-version CC schemes, `MVCC` takes an append-only update approach and maintains a version history, and uses timestamps to determine which version of records to serve. `MVCC` avoids reader-writer conflicts by forwarding them to physically different versions, thereby making this scheme more robust than `OCC` under high contention. To support time traveling and deciding the commit order, `MVCC` relies on logical timestamping, in the read and validation phase, which leads to severe scalability collapse (4.1–31.1× in Figure 3.9) due to the timestamp allocation with increasing core count [100]. We chose `Hekaton` as our example because it is the state-of-the-art in-memory database system [101]. Although it supports multiple CC levels with varying consistency guarantees, we focus on serializable, optimistic MVCC mode. We first describe the workings of the original algorithm and then introduce our modifications with the `Ordo` primitive.

Hekaton has the same three phases as OCC and works as follows: A worker in a transaction, reads the global clock at the beginning of the transaction. During the *read* stage, a transaction reads only a version when its begin timestamp falls between the begin and end timestamps[2] of the committed version. During update, a worker immediately appends its version to the database. The version consists of the transaction ID (TID) of the owner transaction marked in the begin timestamp field. At commit, the transaction assigns a commit timestamp to 1) determine the serialization order, 2) validate the read/scan sets, and 3) iterate the write set to replace TIDs installed in the begin timestamp field with that timestamp. Meanwhile, another transaction that encounters a TID-installed version examines visibility with the begin and end timestamps of the owner transaction.

We apply the Ordo primitive by first replacing the timestamp allocation to use new_time method in the beginning and during the commit phase of a transaction. As a result, this modification introduces uncertainty to compare timestamps from different local clocks during the visibility check. We substitute the comparison with cmp_time() to ensure correctness. Thus, a worker proceeds only if the difference between timestamps is greater than the ORDO_BOUNDARY that provides a definite precedence relation. Otherwise, we either restart that transaction or force it to abort. However, given the small size of ORDO_BOUNDARY, we expect the aborts caused by this uncertainty to be rare. In terms of correctness, our approach provides the same consistency guarantee as the original one, i.e., 1) obtaining a unique timestamp, which new_time() ensures, and 2) maintaining the serial schedule, which is also maintained by cmp_time(), that only, conservatively, commits the transaction that has a precedence relation.

3.2.3 Software Transactional Memory (TL2)

We choose TL2 [1], an ownership- and word-based STM algorithm that employs timestamping for reducing the common-case overhead of validation. TL2 works by ordering

[2]Each version of records have begin timestamp, which indicates when the version becomes valid, and an end timestamp, which denotes when the version becomes invalid.

the update and access relative to a global logical clock and checking the ownership record only once, i.e., it validates all transactional reads at the time of commit. The algorithm works as follows: 1) A transaction begins by storing a global time value ($start$). 2) For a transactional load, it first checks whether an orec[3] is unlocked and has no newer timestamp than the start timestamp of that transaction; it then adds the orec to its read set, and appends the address-value pair in the case of a transactional write. 3) At the time of commit, it first acquires all of the locks in the write set and then validates all elements of the read set by comparing the timestamp of the orec with that of the start timestamp of the transaction. If successful, the transaction writes back the data and obtains a new timestamp, denoting the end of the timestamp (end). It uses the end timestamp as a linearization point, which it atomically writes in each orec of the write set (write address) and also releases the lock. Here, end is guaranteed to be greater than the $start$ timestamp because a transaction atomically increments the global clock, which ensures the linearizability of a transactional update.

The basic requirement of the `TL2` algorithm is that end should be greater than $start$, which the `Ordo` primitive (`new_time()`) ensures. Moreover, two disjoint transactions can share the same timestamp [102]. Thus, by using the `Ordo` API, we modify the algorithm as follows: We assign $start$ with the value of `new_time()`, and use `new_time()` to obtain a definite newer timestamp for the end variable. Here, we again adopt a conservative approach of aborting transactions if two timestamps fall in the `ORDO_BOUNDARY`, as this can corrupt the memory or result in an undefined behavior of the program [1]. Although we can even use timestamp extension to remove aborts that occur during the read timestamp validation at the beginning of a transactional read, it may not benefit us because of the very small `ORDO_BOUNDARY`. Our modification ensures linearizability by 1) first providing an increasing timestamp via `new_time()` that globally provides a new timestamp value and 2) always validating the read set at the time of commit. In addition, we abort transactions if two timestamps (read set timestamp and commit timestamp) fall in the `ORDO_BOUNDARY` to

[3]Orec is a ownership record, which either stores the identity of a lock holder or the most recent unlocked timestamp.

remove uncertainty during the validation phase. Similarly, while performing a transactional load, we apply the same strategy to remove any inconsistencies.

3.2.4 `Oplog`: An Update-heavy Data Structures Library

Besides logical timestamping, physical timestamping is becoming common. In particular, there is an efficient implementation of a concurrent stack [43] and an update-heavy concurrency mechanism (`Oplog` [6]), with the same commonality of performing operations in a decentralized manner. We extend the `Ordo` primitive to `Oplog`. It maintains a per-core log, which stores update operations in a temporal order, and it applies logs on the centralized data structure in a sorted temporal order during the read phase. To ensure the correct temporal ordering across all per-core logs, `Oplog` requires a system-wide synchronized clock to determine the ordering of operations. We modify `Oplog` to use the `new_time()` method, which has a notion of a globally synchronized hardware clock while appending operations to a per-core log and use `cmp_time()` to compare timestamps. `Oplog` ensures linearizability by relying on the system-wide synchronized clock, which ensures that the obtained timestamps are always in increasing order. Our modification guarantees linearizability because `new_time()` exposes a globally synchronized clock, which always provides a monotonically increasing timestamp that will always be greater than the previous value across all invariant clocks. There is a possibility that two or more timestamps fall inside the `ORDO_BOUNDARY` during the merge phase, which denotes concurrent update operations, which is also possible in the original `Oplog` design. We address this problem by using the same technique of the original `Oplog` design, which is to apply these operations in an ascending order of the core ID.

3.3 Implementation

Our library and micro benchmarks comprise 200 and 1,100 lines of code (LoC), in C, respectively, which support architecture-specific timers for different architectures. We modify various programs to show the effectiveness of our `Ordo` primitive. We modify 50

LoC in the RLU code base to support both the Ordo primitive and the architecture-specific code for ARM. To specifically evaluate database concurrency protocols, we choose DBx1000 [103] because it includes all database CC algorithms. We modify 400 LoC to support both OCC and Hekaton algorithms, including changes for the ARM architecture. We use an x86 port of the TL2 algorithm [104], which we extend (25 LoC) using the Ordo API.

3.4 Evaluation

We evaluate the effectiveness of the Ordo primitive by answering the following key questions:

- How does an invariant hardware clock scale on various commodity architectures? (§3.4.1)
- What is the scalability characteristic of the Ordo primitive? (§3.4.2)
- What is the impact of the Ordo primitive on algorithms that rely on synchronized clocks? (§3.4.3)
- What is the impact of the Ordo primitive on version-based algorithms? (§3.4.4, §3.4.5, §3.4.6)
- How does the ORDO_BOUNDARY affect the scalability of algorithms? (§3.4.7)

Experimental setup. Table 3.1 lists the specifications of four machines, namely, a 120-core Intel Xeon (having two hyperthreads), a 64-core Intel Xeon Phi (having four hyperthreads), a 32-core AMD, and a 96-core ARM machine. The first three machines have x86 architecture, out of which Xeon has a higher number of physical cores and sockets, Phi has a higher degree of parallelism, and AMD is from a different processor vendor. These three processors have invariant hardware clocks in their specification. Moreover, we also use a 96-core ARM machine, whose clock is different from existing architectures. It supports a separate generic timer interface, which exists as a separate entity inside a processor [24]. We evaluate the scalability of clocks and algorithms up to the maximum number of hardware threads in a machine.

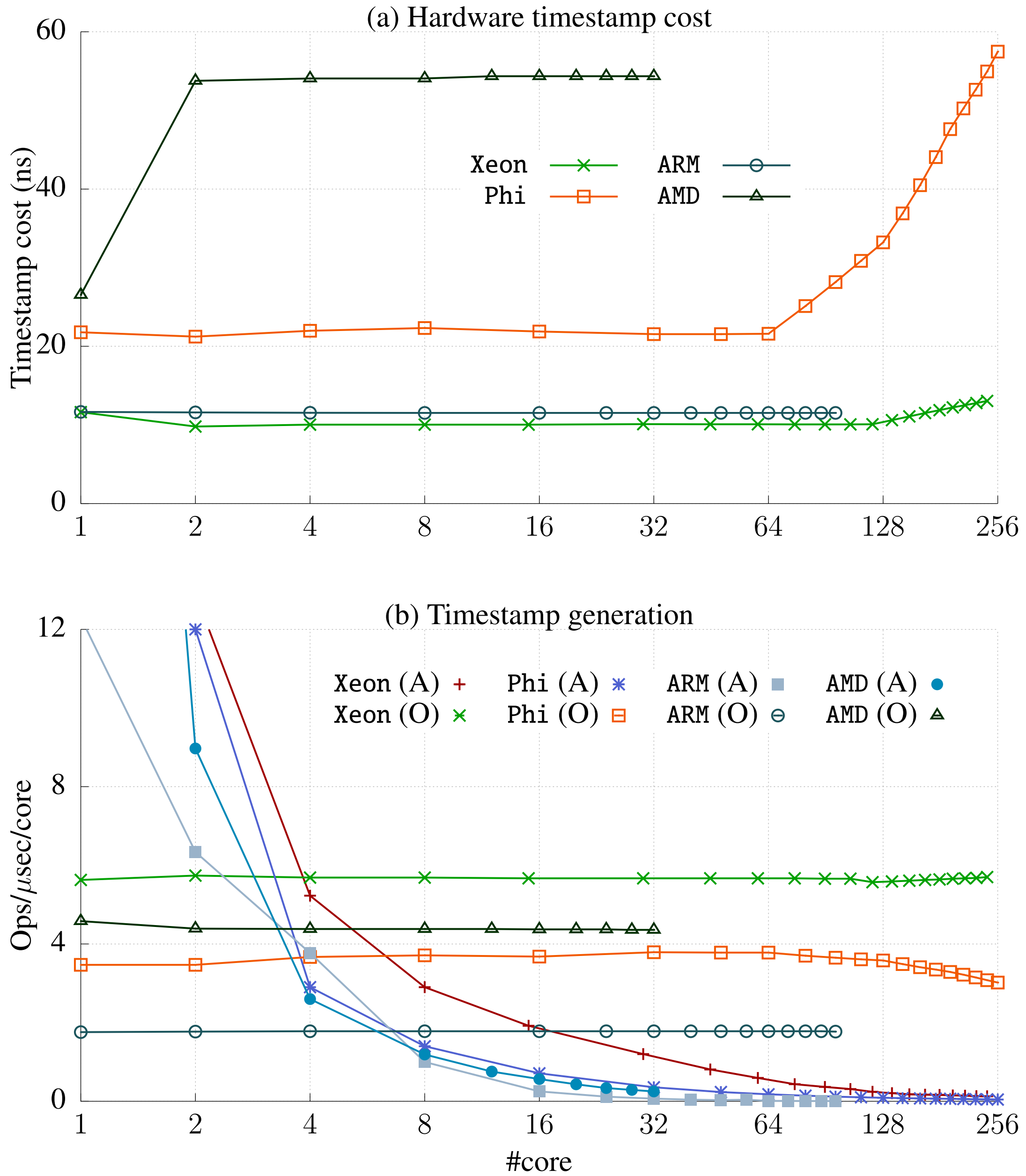

Figure 3.3: Micro evaluation of the invariant hardware clocks used by the `Ordo` primitive. (a) shows the cost of a single timestamping instruction when it is executed by varying the number of threads in parallel; (b) shows the number of per-core generated timestamps in a micro second with atomic increments (**A**) and with `new_time()` (**O**), which generates timestamps at each `ORDO_BOUNDARY`.

Table 3.1: Various machine configurations that we use in our evaluation as well as the calculated offset between cores. While min is the minimum offset between cores, max is the global offset, called ORDO_BOUNDARY (refer to Figure 3.1), which we used, including up to the maximum hardware threads (Cores$*$SMT) in a machine.

Machine	Cores	SMT	Speed (GHz)	Sockets	Offset between clocks min (ns)	max (ns)
Intel Xeon	120	2	2.4	8	70	276
Intel Xeon Phi	64	4	1.3	1	90	270
AMD	32	1	2.8	8	93	203
ARM	96	1	2.0	2	100	1,100

3.4.1 Scalability of Invariant Hardware Clocks

We create a simple benchmark to measure the cost of hardware timestamping instructions on various architectures. The benchmark forks a process on each core and repeatedly issues the timestamp instruction for a period of 10 seconds. Figure 3.3 shows the cost of the timestamp instruction on four different architectures. We observe that this cost remains constant up to the physical core count, but increases with increasing hardware threads, which is evident in Xeon and Phi. Still, it is comparable to an atomic instruction operation in a medium contended case. One important point is that ARM supports a scalable timer whose cost (11.5 ns) is equivalent to that of Xeon (10.3 ns). In summary, the current hardware clocks are a suitable foundation for mitigating contention problem of the global clock with increasing core count, potentially together with hyperthreads.

3.4.2 Evaluating Ordo Primitive

Figure 3.4 presents the measured offset (δ_{ij}) for each pair-wise combination of the cores (with SMT). The heatmap shows that the measured offset between adjacent clocks inside a socket is the least on every architecture. One important point is that all measured offsets are positive. As of this writing, we never encountered a single negative measured offset while measuring the offset in either direction from any core over the course of the past two months. This holds true with prior results [40, 6, 42] and illustrates that the added one-way-delay

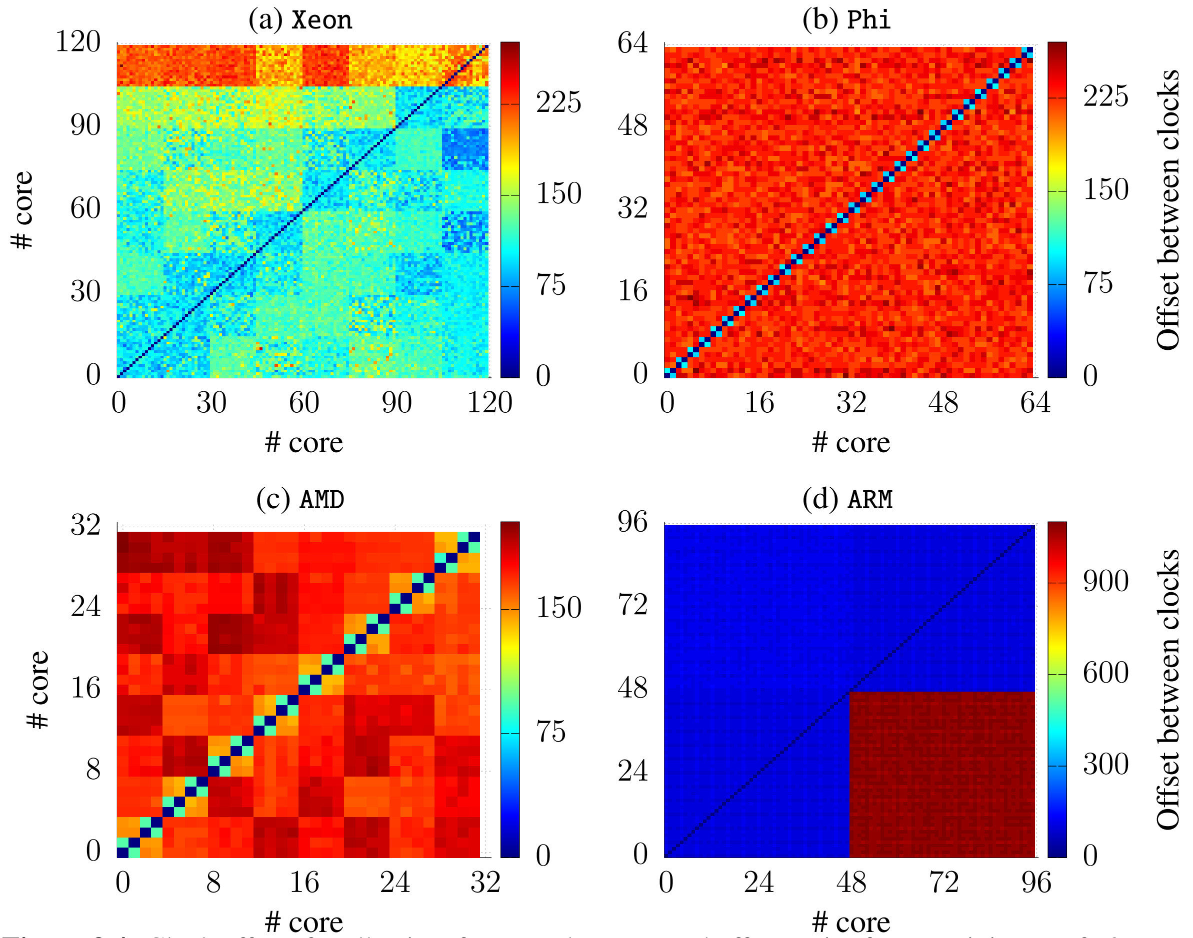

Figure 3.4: Clock offsets for all pairs of cores. The measured offset varies from a minimum of 70 ns to 1,100 ns for all architectures. Both `Xeon` (a) and `ARM` (d) machines show that the one of the sockets has a 4–8× higher offset than the others. To confirm this, we measured the bandwidth between the sockets, which is symmetric in both machines.

latency is greater than the physical offset, which always results in giving a positive measured offset in any direction. For a certain set of sockets, we observe that the measured offset, within that socket is less than the global offset. For example, the fifth socket in the `Xeon` machine has a maximum offset of 120 ns compared with the global offset of 276 ns.

We can define the `ORDO_BOUNDARY` based on an application use in which the application can choose the maximum offset of a subset of cores or sockets inside a machine. But, they need to embed the timestamp along with the core or thread id, which will shorten the length of the timestamp variable and may not be advantageous (§3.4.7). Therefore, we choose the global offset across all cores as the `ORDO_BOUNDARY` for all of our experiments. Table 3.1

shows the minimum and maximum measured offset for all of the evaluated machines, with maximum offset being the ORDO_BOUNDARY. We create a simple micro benchmark to show the impact of timestamp generation by each core by using our new_time() and the atomic increments. Figure 3.3 (b) shows the results of obtaining a new timestamp from a global clock, which is a representative of both physical timestamping and read-only transactions. The Ordo-based timestamp generation remains almost constant up to the maximum core count. It is 17.4–285.5× faster than the atomic increment at the highest core count, thereby showing that transactions that use logical timestamping will definitely improve their scalability with increasing core count.

One key observation is that one of the sockets in both Xeon (eighth socket: 105–119 cores) and ARM (second socket: 48–96 cores) has a 4–8× higher measured offset when measured from a core belonging to the other socket, even though the measured socket bandwidth is constant for both architectures [105]. For example, the measured offset from core 50 to core 0 is 1,100 ns but is only 100 ns from core 0 to core 50 for the ARM machine. We believe that one of the sockets received the RESET signal later than the other sockets in the machine, thereby showing that the clocks are not synchronized. For Phi, we observe that most of the offset lies in the window of 200 ns, but adjacent cores have the least offset.

3.4.3 Physical Timestamping: Oplog

For physical timestamping, we evaluate the impact of Oplog on the Linux reverse map [106]. rmap uses RB-tree data structure for recording the page table entries that map a physical page for every physical page, and it is primarily used by fork(), exit(), mmap(), and mremap() system calls. We modify the reverse mapping for both anonymous and file rmaps in the Linux kernel [6]. We use Exim mail-server [107] on the Xeon machine to evaluate the scalability of the rmap ordo/data structure. Exim is a part of the Mosbench [108] benchmark suite; it is a process-intensive application that listens to SMTP connections and forks a new process for each connection, and a connection forks two more processes to perform

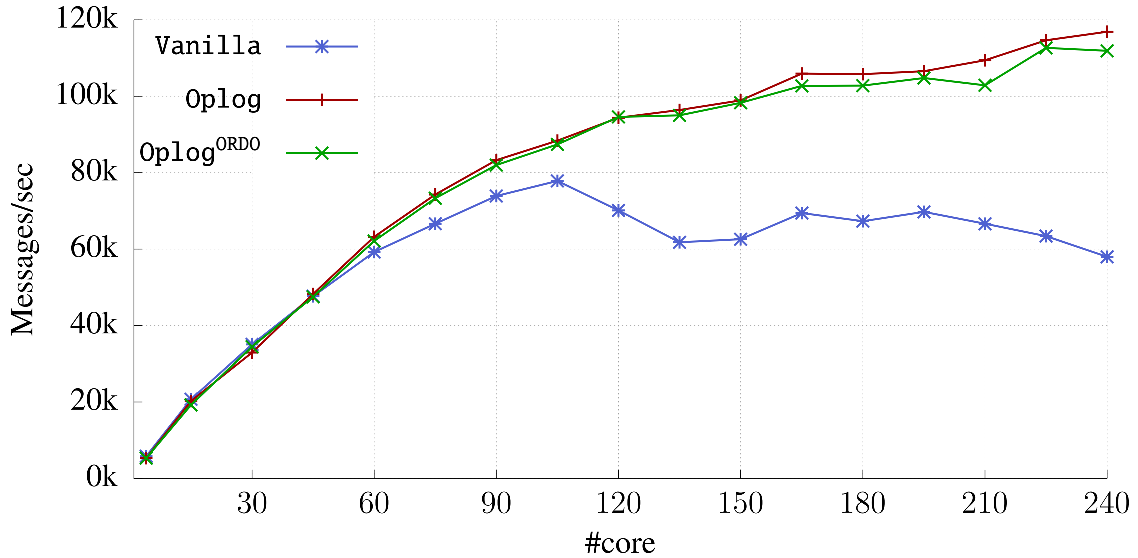

Figure 3.5: Throughput of `Exim` mail-server on a 240-core machine. `Vanilla` represents the unmodified Linux kernel, while `Oplog` is the `rmap` ordo/data structure modified by the `Oplog` API in the Linux kernel. `Oplog`$^{\text{ORDO}}$ is the extension of `Oplog` with the `Ordo` primitive.

various file system operations in shared directories as well as on the files. Figure 3.5 shows the results of `Exim` on the stock Linux (`Vanilla`) and our modifications that include kernel versions with (`Oplog`$^{\text{ORDO}}$) and without the `Ordo` primitive (`Oplog`). The `Oplog` version directly reads the value from the unsynchronized hardware clocks. The results show that `Oplog` does alleviate the contention from the reverse mapping, which we observe after 60 cores, and the throughput of `Exim` increases by 1.9$\times$ at 240 cores. The `Oplog` version is merely 4% faster than the `Oplog`$^{\text{ORDO}}$ approach because `Exim` is now bottlenecked by the file system operations [77] and zeroing of the pages at the time of forking after 105 cores. We do not observe any huge difference between `Oplog` and `Oplog`$^{\text{ORDO}}$ because a reverse mapping is updated only when a system call is issued, which amortizes the cost `ORDO_BOUNDARY` window, besides other virtual file system layer overhead [77].

3.4.4 Read Log Update

We evaluate the throughput of `RLU` for the hash table benchmark and citrus tree benchmark across architectures. The hash table uses one linked list per bucket, and the key hashes

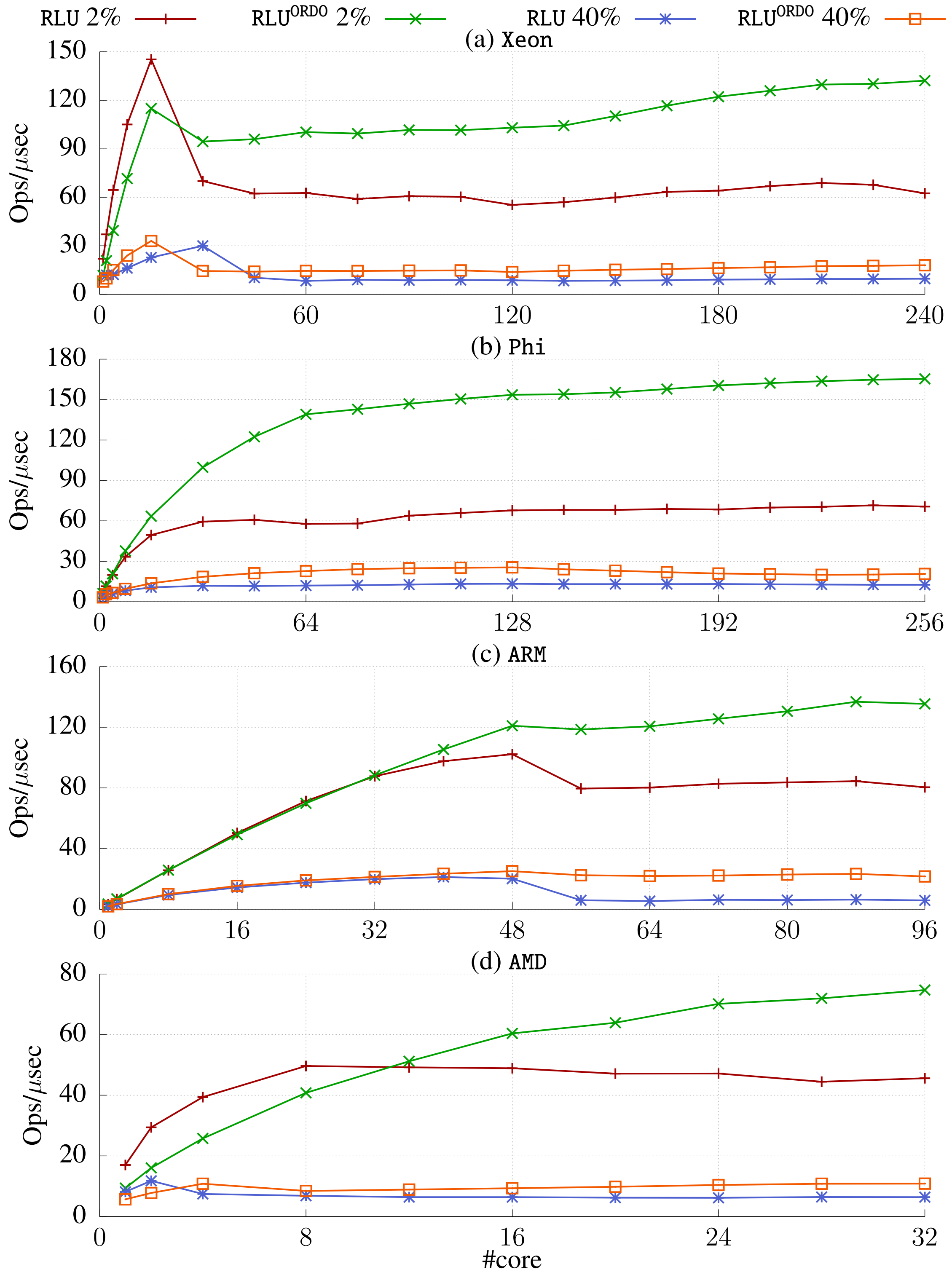

Figure 3.6: Throughput of the hash table with RLU and RLU$^{\text{ORDO}}$ for various update ratios of 2% and 40% updates. The user space hash table has 1,000 buckets with 100 nodes. We experiment it on four machines from Table 3.1.

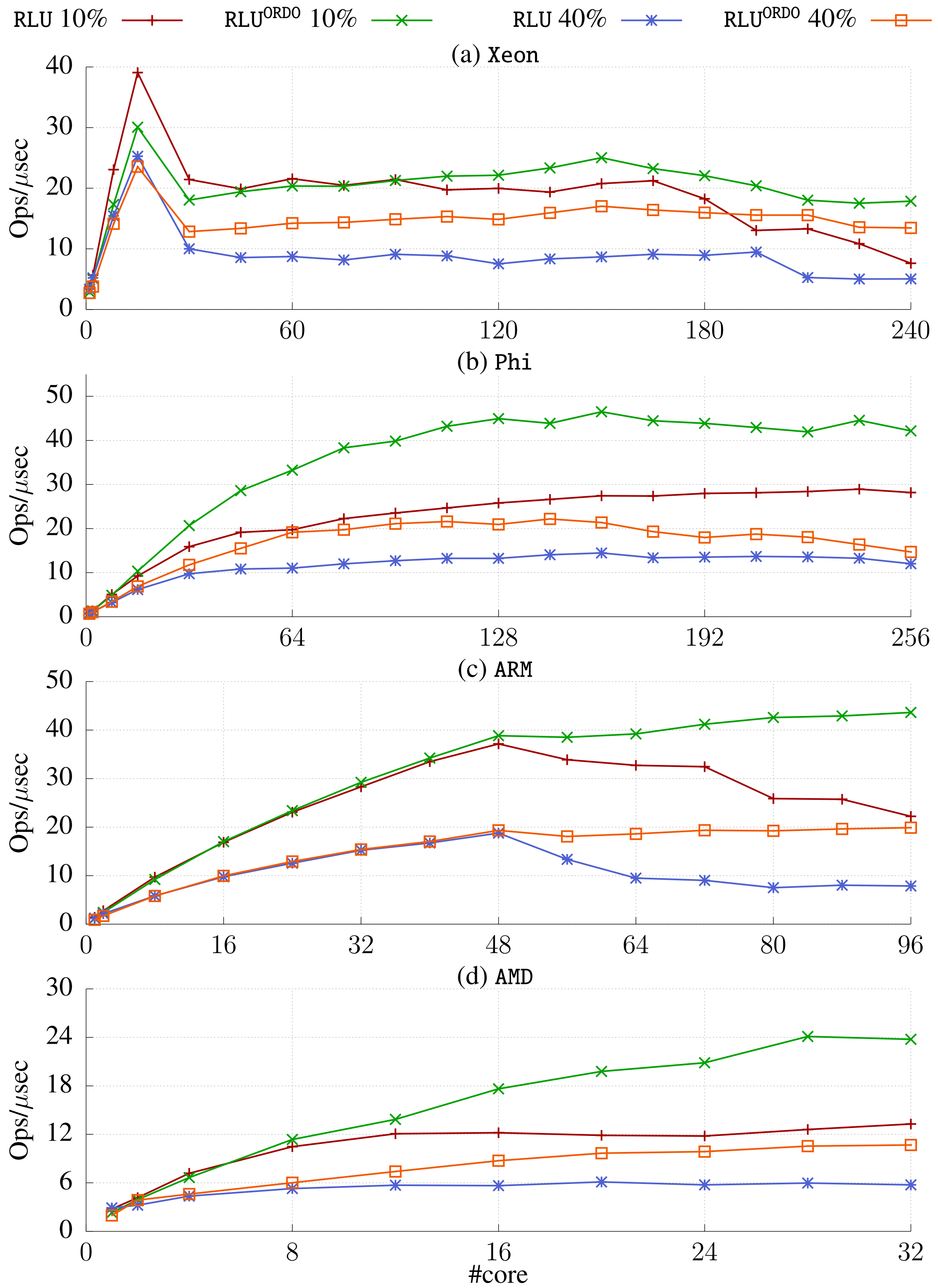

Figure 3.7: Throughput of the citrus tree with RLU and RLU^{ORDO} that consists of 100,000 nodes with varying updates ratio of 10% and 40% on various machines.

into the bucket and traverses the linked list for a read or write operation. Figure 3.6 shows the throughput of the hash table with varying update rates of 2% and 40% across the architectures, where RLU is the original implementation and $\texttt{RLU}^{\texttt{ORDO}}$ is the modified version. The results show that $\texttt{RLU}^{\texttt{ORDO}}$ outperforms the original version by an average of 2.1× across the architectures for all update ratios at the highest core.

Across architectures, the result for the starting number of cores for RLU is better than for $\texttt{RLU}^{\texttt{ORDO}}$ because 1) coherence traffic until certain core counts (Xeon: within a socket, Phi: until 4 cores, ARM: 36 cores and AMD: 12 cores) assists the RLU protocol that has lower abort rates than $\texttt{RLU}^{\texttt{ORDO}}$, and 2) $\texttt{RLU}^{\texttt{ORDO}}$ has to always check for the locks while dereferencing because the invariant clock does not provide the semantic of fetch_and_add(). Thus, in the case of only readers (100% reads), $\texttt{RLU}^{\texttt{ORDO}}$ is 8% slower than RLU at the highest core across the architectures, but even with a 2% update rate, it is the atomic update that degrades the performance of RLU. On further analysis, we find that the $\texttt{RLU}^{\texttt{ORDO}}$ spends at most 20% less time in the synchronize phase, which happens because of the decrease in the cache-coherence traffic on higher core count across all architectures. Furthermore, our Ordo's new_time() does not act as a backoff mechanism. Figure 3.12 illustrates that even on varying the ORDO_BOUNDARY boundary by 1/8×–8×, the scalability of RLU algorithm changes by only ±3% while running on 1-core, 1-socket, and 8-sockets.

Overall, all multisocket machines show a similar scalability trend after crossing the socket boundary and are saturated after a certain core count because they are bottlenecked by the locking and creation of the object and its copy, which is more severe in the case of ARM for crossing the NUMA boundary, as is evident after 48 cores. In the case of Phi, there is no scalability collapse because it has a higher memory bandwidth and slower processor speed, which only saturates the throughput. However, as soon as we remove the logical clock, the throughput increases by an average of 2× in each case. Even though the cost of the timestamp instruction increases with hyperthreads (3× at 256 threads), the throughput is almost saturated because of the object copying and locking. Even with the deferrals (refer

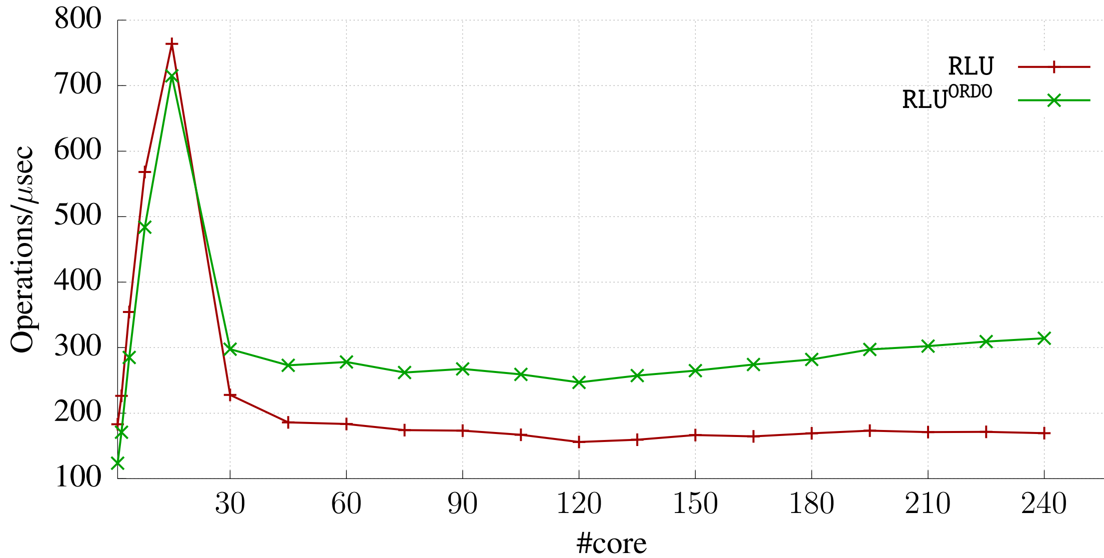

Figure 3.8: Throughput of the hash table benchmark (40% updates) with the deferred-based RLU and RLU$^{\text{ORDO}}$ approach on the Xeon machine. Even with the deferred-based approach, the cost of the global clock is still visible after crossing the NUMA boundary.

to Figure 3.8), RLU$^{\text{ORDO}}$ is at most 1.8× faster than the base version, thereby illustrating that the defer-based approach still suffers from the contention on the global clock. We also evaluate the citrus-tree benchmark that involves complex update operations.

Figure 3.7 shows the results of the citrus-tree benchmark for the 10% and 40% updates across various operations. We can observe that RLU$^{\text{ORDO}}$ outperforms RLU even in the complicated scenario by a factor of two on every architecture. The scalability behavior is similar to the hash table benchmark across the architectures.

3.4.5 Concurrency Control Mechanism

We evaluate the impact of Ordo primitive on the existing OCC and Hekaton algorithms with YCSB [109] and TPC-C benchmarks. We execute read-only transactions to focus only on the scalability aspect without transaction contentions (Figure 3.9), which comprises two read-queries per transaction and a uniform random distribution. We also present results from the TPC-C benchmark with 60 warehouses as a contentious case (Figure 3.10). Figure 3.9 shows that both OCC$^{\text{ORDO}}$ and Hekaton$^{\text{ORDO}}$ not only outperform OCC (5.6–39.7×) and Hekaton

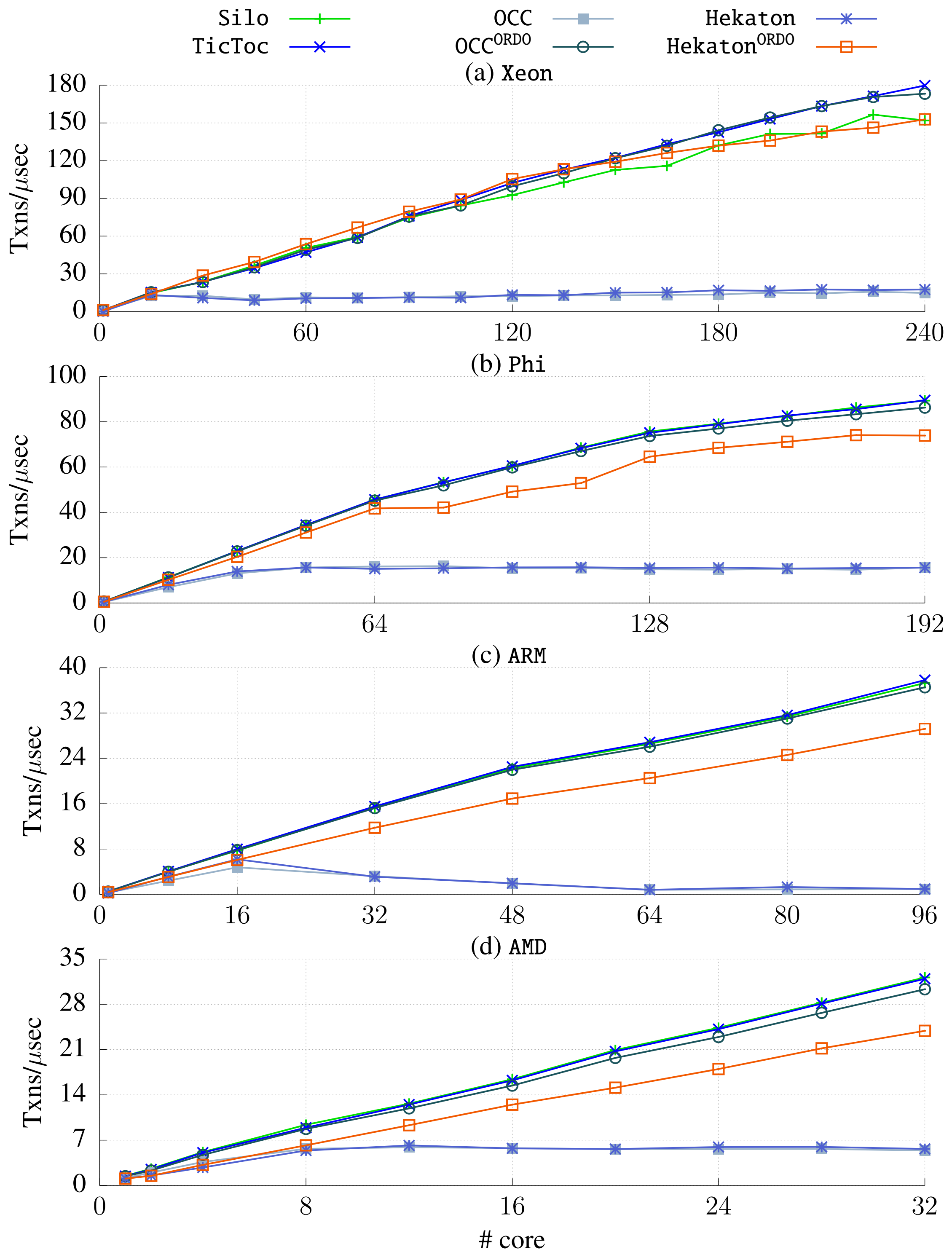

Figure 3.9: Throughput of various concurrency control algorithms of ordo/databases for read-only transactions (100% reads) using the YCSB benchmark on various architectures. We modify the existing `OCC` and `Hekaton` algorithms to use our `Ordo` primitive ($\texttt{OCC}^{\texttt{ORDO}}$ and $\texttt{Hekaton}^{\texttt{ORDO}}$, respectively) and compare them against the state-of-the-art `OCC` algorithms: `Silo` and `TicToc`. Our modifications removes the logical clock bottleneck across various architectures.

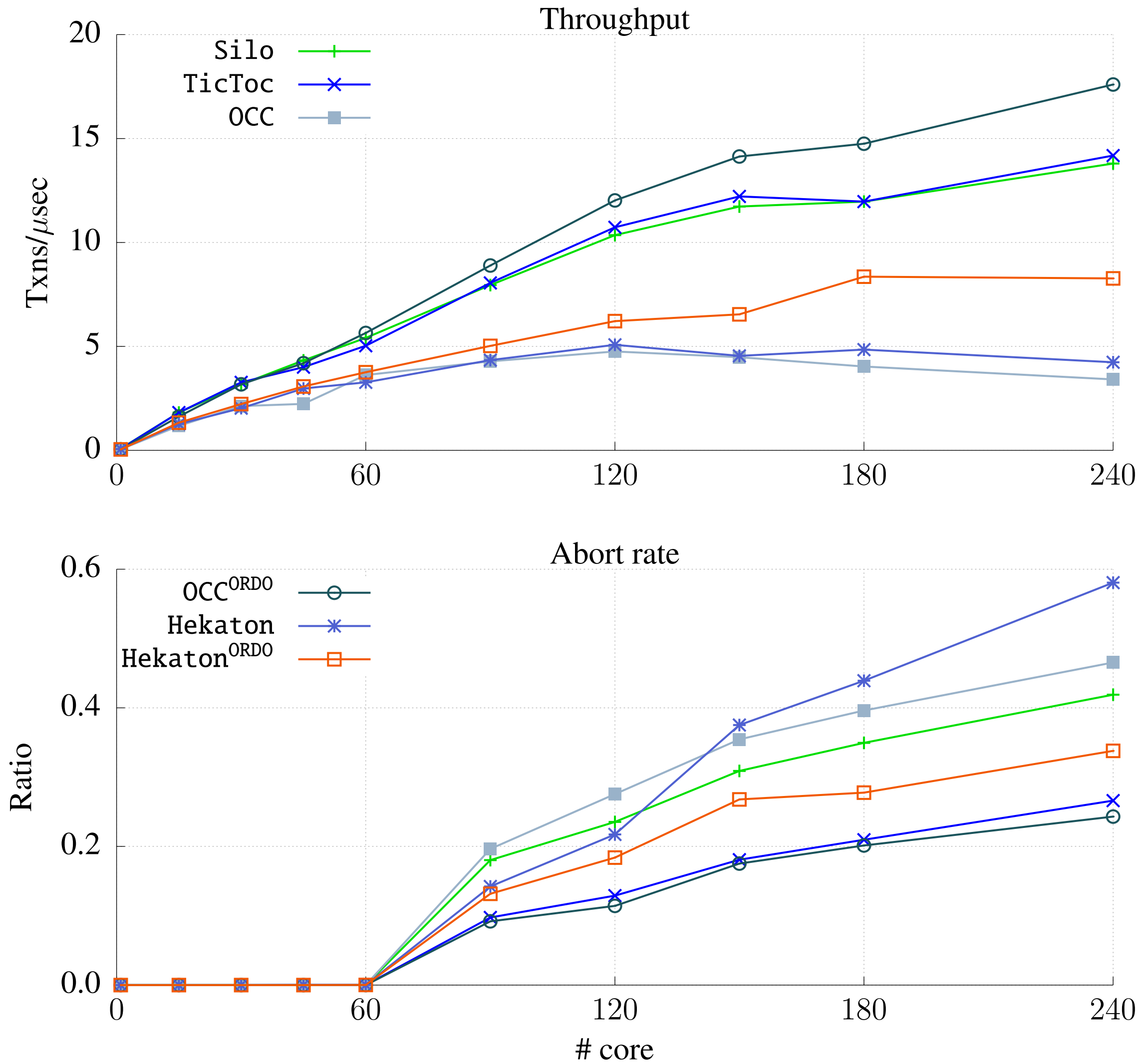

Figure 3.10: Throughput and abort rates of concurrency control algorithms for TPC-C benchmark with 60 warehouses on 240 Intel Xeon machine.

(4.1–31.1×), respectively across architectures, but also achieve almost similar scalability as that of TicToc and Silo, which do not have global timestamp allocation overhead. The reason for such an improvement is that both OCC and Hekaton waste 62–80% of their execution time in allocating the timestamps, which has also been shown by Yu *et al.* [100], whereas Ordo successfully eliminates the logical timestamping overhead with a modest amount of modification. Compared with state-of-the-art optimistic approaches that avoid

global logical timestamping, OCC^{ORDO} shows comparable scalability, thereby enabling Ordo to serve as a push-button accelerator for timestamp-based applications, which provides a simpler way to scale these algorithms. In addition, Hekaton^{ORDO} has comparable performance to that of other OCC-based algorithms and is only 1.2–1.3× slower because of its heavyweight dependency-tracking mechanism that maintains multiple versions.

Figure 3.10 presents the throughput and abort rate for the TPC-C benchmark. We run NewOrder (50%) and Payment (50%) transactions only with hash index. The results show that OCC^{ORDO} is 1.24× faster than TicToc and has a 9% lower abort rate, since TicToc starts to spend more time (7%) in the validation phase because of the overhead of its ordo/data-driven timestamp computation, as it has to traverse the read and write set to find the common commit timestamp; OCC^{ORDO} already has a notion of global time, which speeds up the validation process, thereby increasing its throughput with lower aborts. Thus, hardware clocks provide better scalability than software bypasses that come at the cost of extra computation. In the case of multi-version CC, Hekaton^{ORDO} also outperforms Hekaton by 1.95× with lower aborts.

3.4.6 Software Transactional Memory

We evaluate the impact of Ordo primitive by evaluating the TL2 and TL2^{ORDO} algorithms on the set of STAMP benchmarks. Figure 3.11 presents the speedup over the sequential execution of these benchmarks. The results show that TL2^{ORDO} improves the throughput of every benchmark. TL2^{ORDO} has higher speedup over TL2 for both Ssca2 and Kmeans because they have short transactions, which in turn results in more clock updates. Genome, on the other hand, is dominated by large read conflict-free transactions; thus, it does not severely stress the global clock with an increasing core count. In the case of Intruder, we observe some performance improvement up to 60 cores. However, after 60 cores, TL2^{ORDO} has 10% more aborts than TL2, which in turn slows down the performance of the TL2^{ORDO}, as the bottleneck shifts to the large working set maintained by the basic TL2 algorithm. We can circumvent

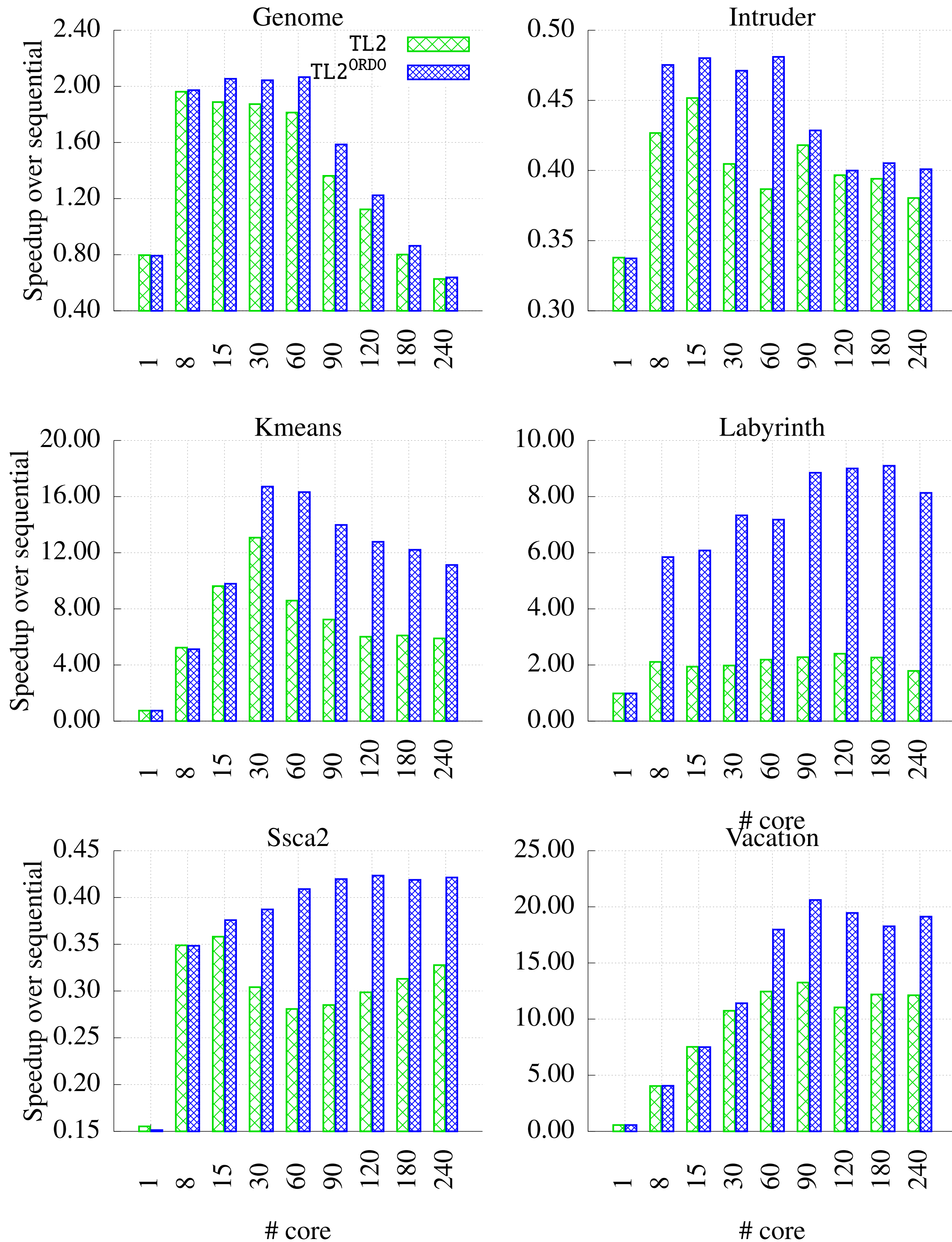

Figure 3.11: Speedup of the STAMP benchmark with respect to the sequential execution on `Xeon` machine for `TL2` and $\texttt{TL2}^{\texttt{ORDO}}$ algorithms. $\texttt{TL2}^{\texttt{ORDO}}$ improves the throughput up to 3.8× by alleviating the cache-line contention that occurs of the global logical clock. $\texttt{TL2}^{\texttt{ORDO}}$ shows significant improvement in the case of workloads running with very short transactions (Kmeans and Ssca2) and the ones with very long transactions by decreasing their aborts due to the cache-line mitigation (Labyrinth).

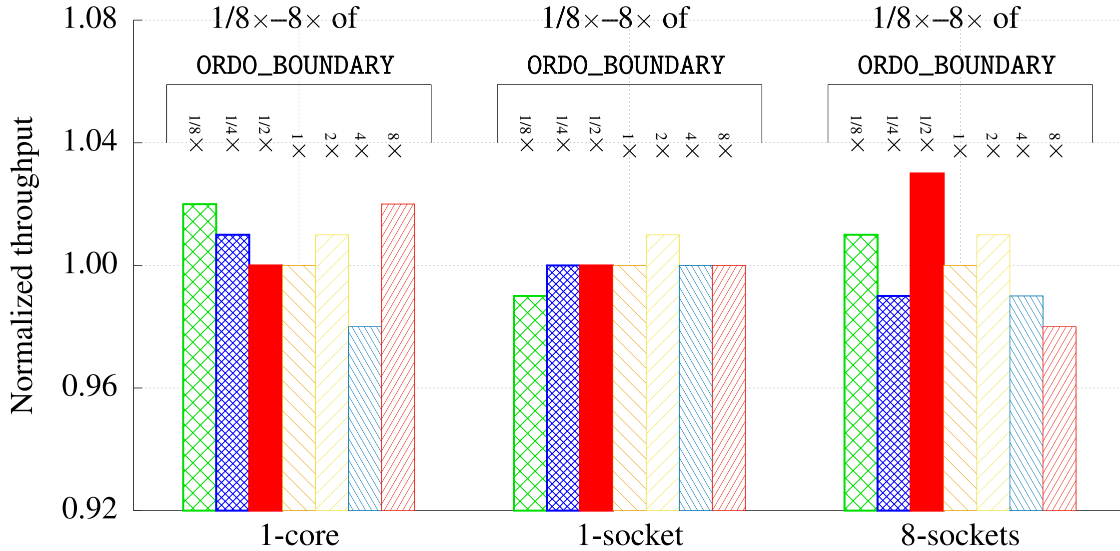

Figure 3.12: Normalized throughput of the RLU$^{\text{ORDO}}$ algorithm for varying ORDO_BOUNDARY on the Xeon machine on 1-core, 1-socket (30 cores), and 8-sockets (240 cores) with 98% reads and 2% writes. We vary the ORDO_BOUNDARY from 1/8×–8× for all three configurations, which shows that the throughput varies by only ±3%, thereby proving two points: 1) timestamping is one of the bottlenecks, and 2) ORDO_BOUNDARY does not act as a backoff mechanism for such logical timestamping-based algorithms.

this problem by employing type-aware transactions [110]. In the case of Labyrinth, we observe that the TL2$^{\text{ORDO}}$ improves the throughput by 2–3.8×. Although Labyrinth has long running transactions [104], we observe that the number of aborts decreases by 5.8–15.5×, which illustrates that transactions in Labyrinth spend almost twice the amount of time in re-executing the aborted transactions, which occurs because of the cache-line contention of the global clock. Finally, TL2$^{\text{ORDO}}$ also improves the performance of Vacation because it is a transactional-intensive workload, as it performs at least 3 M transactions with abort rates on the order of 300K–400K, which results in stressing the global clock. In summary, Ordo primitive improves the throughput of the TL2 by a maximum factor of two.

3.4.7 Sensitivity Analysis of ORDO_BOUNDARY

To show that the calculated ORDO_BOUNDARY does not hamper the throughput of the algorithm, we ran the hash table benchmark with the RLU$^{\text{ORDO}}$ algorithm on the Xeon machine with 98% reads and 2% writes for three configurations: 1-core, 1-socket (30 cores), and 8-sockets

(240 cores). We found that on either increasing or decreasing the `ORDO_BOUNDARY` by 1/8× or up to 8× of the global offset (refer to Table 3.1), the scalability of $\texttt{RLU}^{\texttt{ORDO}}$ algorithm only varies by ±3% in all three configurations because the algorithm has other bottlenecks that overshadow the cost of both the lower and higher offsets. We can further confirm this result from Figure 3.3 (b) results, as hardware clocks can generate 1.4 billion timestamps at 240 cores, while the maximum throughput of the hash table is only 102 million for a 2% update operation. We observe a similar trend on a single core as well as for multiples of sockets. Furthermore, we can also infer that another important point is that the `Ordo`'s `new_time()` method does not act as a backoff mechanism, as we can see consistent throughput even on decreasing the `ORDO_BOUNDARY` to 1/8×, which is merely 34 ns on the `Xeon` machine. This point also holds true for other timestamp algorithms, as they have to perform other tasks [100, 104].

3.5 Chapter Summary

Ordering still remains the basic building block to reason about the consistency of operations in a concurrent application. However, ordering comes at the cost of expensive atomic instructions that do not scale with increasing core count and further limit the scalability of concurrency on large multicore and multisocket machines. We propose `Ordo`, a simple scalable primitive that addresses the problem of ordering by providing an illusion of a globally synchronized hardware clock with some uncertainty inside a machine. `Ordo` relies on invariant hardware clocks that are guaranteed to be increasing at a constant frequency but are not guaranteed to be synchronized across the cores or sockets inside a machine, which we confirm for `Intel` and `ARM` machines. We apply the `Ordo` primitive to several timestamp-based algorithms, which use either physical or logical timestamping, and scale these algorithms across various architectures by at most 39.7×, thereby making them multicore friendly.

CHAPTER 4
IMPACT OF SCHEDULING ON VIRTUAL MACHINES

Virtualization is now the backbone of every cloud-based organization to run and scale applications horizontally on demand. Recently, this scalability trend is also extending towards vertical scaling [111, 112], *i.e.*, a virtual machine (VM) has up to 128 virtual CPUs (vCPUs) and 3.8 TB of memory to run large in-memory databases [18, 19] and data processing engines [20]. At the same time, cloud providers strive to oversubscribe their resources to improve hardware utilization and reduce energy consumption, without imposing any permissible overhead on the application [113, 114]. Thus, the multiplexing of VMs leads to a breed of different types of symptoms in the form preemption problems that occur because of the double scheduling problem [115]: 1) the guest OS schedules processes on vCPUs and 2) the hypervisor schedules vCPUs on physical CPUs. The root cause of this double scheduling phenomenon is a semantic gap between a hypervisor and guest OSes, in which the hypervisor is agnostic of not only the scheduling of VMs but also guest OS-specific critical code that deter the scalability of applications. Some of the prior works address this problem by adopting co-scheduling approaches [116, 117, 118], which can suffer from priority inversion, CPU fragmentation, and may mitigate the double scheduling symptoms [115]. Such symptoms, that have mostly been addressed individually, are lock-holder preemption (LHP) [84, 83, 86, 119], lock-waiter preemption (LWP) [84], and blocked-waiter wakeup (BWW) [120, 16], problems.

To solve these symptoms as a whole, we make a key observation: These symptoms occur because 1) the hypervisor is scheduling out a vCPU at a time when the vCPU is executing a critical code, and 2) a vCPU, waiting to acquire a lock, is either uncooperative or sleeping [66], leading to LWP and BWW issues. Thus, we propose an alternative perspective, *i.e.*, instead of devising a solution for each symptom, we use four key ideas that allows a VM to hint the

hypervisor for making an effective scheduling decision to allow its forward progress. First, we consider all of the locks and interrupt contexts as critical components. Second, we devise a set of para-virtualized methods that annotate these critical components as *enlightened critical sections* (eCS). These methods are lightweight in nature and notify a hypervisor from the VM and vice-versa with memory operations via shared memory, while avoiding the overhead of hypercall and interrupt injection. Third, the hypervisor now can figure out whether a vCPU is executing an eCS and can reschedule it. However, by rescheduling a vCPU, we introduce unfairness in the system. We tackle this issue with the OS's fair scheduling policy [82], which compensates for that additional schedule by allowing other tasks to run for extra time, thereby maintaining the eventual fairness in the system. Lastly, we leverage our methods to design a virtualized schedule-aware spinning strategy that enables lock waiters to be work conserving as well as cooperative inside a VM. That is, a vCPU now cooperatively spins for the lock, if a physical CPU is under-committed, else it yields the vCPU.

Our approach improves the scalability of real-world applications by 1.2–1.6× in an under-committed case. Moreover, our eCS annotation, combined with eSCHDSPIN, avoids preemption by 85–100%, while improving the scalability of applications by 1.4–2.5× in an over-committed scenario on an 80-core machine.

4.1 Design

A hypervisor can mitigate various preemption problems, if it is aware of a vCPU executing a critical section. We denote such a hypervisor-aware critical section as an *enlightened critical section* (eCS), that can be executed for one more schedule. eCS is applicable to all synchronization primitives and mechanisms such as RCU and interrupt contexts. We now present our lightweight methods that act as a cross-layer interface for annotating an eCS and later focus on our notion of an extra schedule and our approach to maintain eventual fairness in the system.

Table 4.1: Set of para-virtualized methods exposed by the hypervisor to a VM for providing hints to the hypervisor to mitigate double scheduling. These methods provide hints to the hypervisor and VM via shared memory. A vCPU relies on the first four methods to ask for an extra schedule to overcome LHP, LWP, RP, RRP, and ICP. Meanwhile, a vCPU gets hints from the hypervisor by using the last two methods to mitigate LWP and BWW problems. The cpu_id is the core id that is used by tasks running inside a guest OS.
†Currently, is_vcpu_preempted() is already exposed to the VM in Linux.

Hint	Lightweight Para-virtualized API	Description
VM → Hypervisor	void activate_non_preemptable_ecs(cpu_id)	Increase the eCS count for a vCPU with cpu_id by 1 for a non-preemptable task
	void deactivate_non_preemptable_ecs(cpu_id)	Decrease the eCS count for a vCPU with cpu_id by 1 for a non-preemptable task
	void activate_preemptable_ecs(cpu_id)	Increase the eCS count for a vCPU with cpu_id by 1 for a preemptable task
	void deactivate_preemptable_ecs(cpu_id)	Decrease the eCS count for a vCPU with cpu_id by 1 for a preemptable task
Hypervisor → VM	bool is_vcpu_preempted(cpu_id)†	Return whether a vCPU with cpu_id is preempted by the hypervisor
	bool is_pcpu_overcommitted(cpu_id)	Return whether a physical CPU, running a vCPU with cpu_id, is over-committed

4.1.1 Lightweight Para-virtualized methods

We propose a set of six *lightweight para-virtualized methods* to bridge the semantic gap that both VM and hypervisor use for conveying information between them. These methods rely on four variables (refer Figure 4.1) that are *local* to each vCPU. They are exposed via shared memory between the hypervisor and a VM and the notification happens via simple read and write memory operations. A simple memory read is sufficient for the hypervisor to decide on scheduling because 1) it tries to execute each vCPU on a separate pCPU, 2) and it requires knowing about an eCS only at the schedule boundary, thereby removing the cost of polling and other synchronous notifications [121]. To consider an OS critical section as an eCS, we mark the start and unmark the end of a critical section, which lets the hypervisor know about an eCS. However, a process in an OS can be of two types. First is the non-preemptable process that can never be scheduled out. Such a process is either an interrupt or a kernel thread running after acquiring a spinlock. Another one is the preemptable task such as a user process or a process with blocking lock. Hence, we introduce four methods (VM $\rightarrow$ Hypervisor) to separately handle these two types of tasks. The last two methods (Hypervisor $\rightarrow$ VM) provide the hypervisor context to the VM, which a lock waiter can use to mitigate the LWP problem or yield the vCPU to other hypervisor tasks or vCPUs in an over-committed scenario. Figure 4.1 illustrates those four states:

- **non_preemptable_ecs_count** maintains the count of active non-preemptable eCSs, such as non-blocking locks, RCU reader, and interrupt contexts. It is similar to the preemption count of the OS.
- **preemptable_ecs_count** is similar to the preemption count variable of the OS, but it only maintains the count of active preemptable eCSs, such as blocking primitives, namely, mutex and rwsem.
- **vcpu_preempted** denotes whether a vCPU is running. It is useful for handling the BWW problem in both under- and over-committed scenarios.
- **pcpu_overloaded** denotes whether a physical CPU, executing that particular vCPU, is

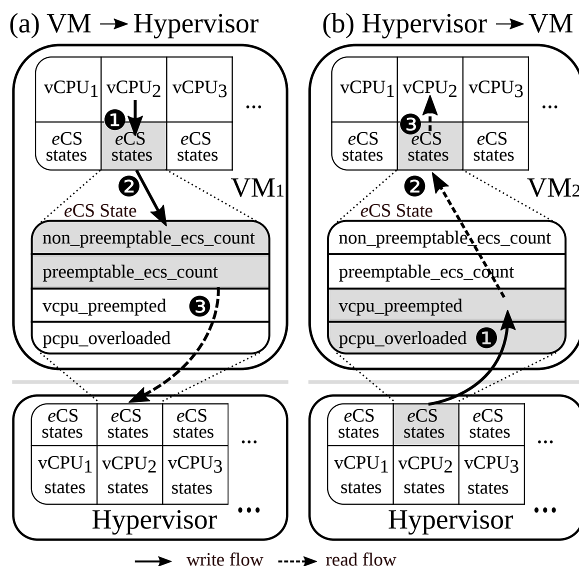

Figure 4.1: Overview of the information flow between a VM and a hypervisor. Each vCPU has a per-CPU state that is shared with the hypervisor, denoted as *e*CS state. Figure (a) shows how the $vCPU_2$ relays information about an *e*CS to the hypervisor. On entering a critical section or an interrupt context (❶), $vCPU_2$ updates the non_preemptable_ecs_count (❷). After a while, before scheduling out $vCPU_2$, the hypervisor reads its *e*CS state (❸), and allows it run for one more schedule to mitigate any of the double scheduling problems. Figure (b) shows how the hypervisor shares the information whether a vCPU is preempted or a physical CPU is overloaded, at the schedule boundary. For instance, the hypervisor marks vcpu_preempted, while scheduling out a vCPU; or updates pcpu_overloaded flag to one if the number of active tasks on that physical CPU is more than one. Both try to further mitigate LWP and BWW problems.

over-committed. Lock waiters can use this information to address the BWW problem in an over-committed scenario.

Figure 4.1 presents two scenarios in which the schedule context information is shared between a vCPU and the hypervisor. Figure 4.1 (a) shows how a vCPU, *i.e.*, entering an *e*CS, shares information with the hypervisor. During entry (❶), $vCPU_2$ first updates its corresponding state (non_preemptable_ecs_count or preemptable_ecs_count) (❷) and continues to execute its critical section. Meanwhile, the hypervisor, before scheduling out $vCPU_2$, checks $vCPU_2$'s *e*CS states (❸) and allows it to run for extra time if certain criteria are fulfilled (§4.1.2); otherwise, it schedules out $vCPU_2$ with other waiting tasks. When $vCPU_2$ exits the *e*CS, it decreases the *e*CS state count, denoting the end of critical section. Figure 4.1 (b) illustrates another scenario that addresses the BWW problem. in which the hypervisor updates the *e*CS states: pcpu_overloaded and vcpu_preempted while scheduling in and

out $\texttt{vCPU}_2$, respectively, at each schedule boundary (❶. We devise a simple approach—virtualized scheduling-aware spinning (eSCHDSPIN)—that enables efficient scheduling aware waiting for both blocking and non-blocking locks (§4.2). That is, $\texttt{vCPU}_2$ reads both states (❷ and decides whether to keep spinning until the lock is acquired if the pCPU is not overloaded (❸, else it yields, which allows the other vCPU (in $\texttt{VM}_2$) or a hypervisor's task to progress forward by doing some useful task, thereby mitigating the double scheduling problems.

4.1.2 Eventual Fairness with Selective Scheduling

As mentioned before, the hypervisor relies on its scheduler to figure out whether a vCPU is executing an eCS. That is, when a vCPU with a marked eCS is about to be scheduled out, the hypervisor scheduler checks the value of eCS count variables (Figure 4.1). If any of these values are greater than zero, the hypervisor lets the vCPU run for an extra schedule. However, vCPU rescheduling introduces two problems: 1) How does the hypervisor handles a task with eCS, which the guest OS can preempt or schedule out? 2) How does it ensure the system fairness?

We handle an eCS preemptable task with `preemptable_ecs_count` counter methods, which differentiate between a preemptable task and a non-preemptable task. We do so because the guest OS can schedule out a preemptable task. In this case, the hypervisor should avoid rescheduling that vCPU because 1) it will result in false rescheduling, and 2) it can hamper the VM performance. We address this issue inside the guest OS, *i.e.*, before scheduling out an eCS-marked task inside a guest OS, we save the value of `preemptable_ecs_count` to a task-specific structure and reset the counter to zero. Later, when the task is rescheduled again by the guest OS, we restore the `preemptable_ecs_count` with the saved value from the task-specific structure, thereby mitigating the false scheduling.

With vCPU rescheduling, we introduce unfairness at two levels: 1) An eCS marked vCPU

Table 4.2: Applicability of our six lightweight para-virtualized methods that strive to address the symptoms of double scheduling.

Method	LHP	RP	RRP	ICP	LWP	BWW
activate_non_preemptable_vcs()	✓	✓	✓	✓	-	-
deactivate_non_preemptable_vcs()	✓	✓	✓	✓	-	-
activate_preemptable_vcs()	✓	✓	-	-	-	-
deactivate_preemptable_vcs()	✓	✓	-	-	-	-
is_vcpu_preempted()	-	-	-	-	✓	✓
is_pcpu_overcommitted()	-	-	-	-	-	✓

will always ask for rescheduling on every schedule boundary.[1] 2) By rescheduling a vCPU, the hypervisor is unfair to other tasks in the system. We resolve the first issue by allowing the hypervisor to reschedule an eCS-marked vCPU only once during that schedule boundary as rescheduling extends the boundary. At the end of schedule boundary, the hypervisor schedules other tasks to avoid the starving other tasks or VMs and addresses indefinite rescheduling. In addition, the hypervisor also keeps track of this extra reschedule information and runs other vCPUs for longer duration and inherently balances the running time, an equivalent to vCPU penalization. Thus, our approach selectively reschedules and penalizes a vCPU rather than balancing the extra reschedule information across all cores, which will result in an unnecessary overhead of synchronizing all runtime information of rescheduling. We call our approach as the *local CPU penalization* approach, as we only penalize a vCPU that executed an eCS, thereby ensuring *eventual fairness* in the system. Moreover, our local vCPU scheduling is a form of selective-relaxed co-scheduling of vCPUs depending on what kind of tasks are being executed, while without maintaining any synchronization among vCPUs, unlike prior approaches [116, 117].

4.2 Use Case

The double scheduling phenomenon introduces the semantic gap in three places: 1) from a vCPU to a physical CPU that results in LHP, RP, and ICP problems; 2) from a pCPU to a vCPU; and 3) from one vCPU to another in a VM, both suffer from LWP and BWW problems. Table 4.2

[1] Such a VM can be either an I/O or an interrupt-intensive VM that spends most of its time in the kernel, or even a compromised VM.

shows how to use our methods to address these problems.

LHP, RP, RRP, and ICP problem. To circumvent these problems, we rely on the VM → hypervisor notification because a vCPU running any spinlocks, read-write locks, mutex, rwsem, or an interrupt context is already inside the critical section. Thus, we call activate_*() and deactivate_*() methods for annotating critical sections. For example, the first two methods are applicable to spinlocks, read-write locks, RCU, and interrupts, and the next two are for mutex and rwsem. (refer Table 4.2).

LWP and BWW problem. The LWP problem occurs in the case of FIFO-based locks such as MCS and Ticket locks [59]. However, unfair locks, such as qspinlock [88], mutex [122], and rwsem [76], do not suffer from this problem, and are currently used in Linux. The reason is that they allow other waiters to steal the lock, while suffering from the issue of starvation. On the other hand, all of these locks suffer from the BWW problem because the cost to wake up a sleeping in a virtualized environment varies from 4,000–10,000 cycles. as a wake-up call results in a VMexit, which adds an extra overhead to notify a vCPU to wake up a process. This problem is severe for blocking primitives because they are non-work conserving in nature [66], *i.e.*, the waiters schedule out themselves, even if a single task is present in the run queue of the guest OS. We partially mitigate this issue by allowing the waiters to spin rather than sleep if a single task is present in the run queue of the guest scheduler (SCHDSPIN). However, this approach is non-cooperative when multiple VMs are running. Thus, to avoid unnecessary spinning of waiters, we rely on our is_pcpu_overcommitted() API that notifies a waiter to only spin if the pCPU is not over-committed. We call this approach the virtualized scheduling-aware spinning approach (*e*SCHDSPIN).

4.3 Implementation

We realized the idea of *e*CS by implementing it on the Linux kernel version 4.13. Besides annotating various locks and interrupt contexts with *e*CS, we specifically modified the scheduler and the para-virtual interface of the KVM hypervisor. Our changes are portable

Table 4.3: *e*CS requires small modifications to the existing Linux kernel, and the annotation effort is also minimal: 60 LoC changes to support the 10 million LoC Linux kernel that has around 12,000 of lock instances with 85,000 lock invocations.

Component	Lines of code
*e*CS annotation	60
*e*CS infrastructure	800
Scheduler extension	150
Total	1,010

enough to apply on the Xen hypervisor too. The whole modification consists of 1,010 lines of code (see Table 4.3).

Lightweight para-virtualized methods. We share the information between the hypervisor and a VM with a shared memory between them, which is similar to the `kvm_steal_time` [123] implementation. For instance, each VM maintains a per-core *e*CS states, and the hypervisor maintains per-vCPU *e*CS states for each VM.

Scheduler extension. We extend a scheduler-to-task notification mechanism, `preempt_noti-fier` [124], for identifying an *e*CS-marked vCPU at the schedule boundary. Our extension allows the scheduler to know about the task scheduling requirement and decide scheduling strategy at the schedule boundary. For example, in our case, the extension reads the `non_preemptable_ecs_count` and `preemptable_ecs_count` to decide the scheduling strategy for the vCPU. Besides this, we rely on the notifier's in and out methods to set the value of `vcpu_preempted` and `pcpu_overloaded` variables.

We implemented our vCPU rescheduling decision in the `schedule_tick` function [125]. The `schedule_tick` function performs two tasks: 1) It does the bookkeeping of the task runtime, which is used for ensuring the fairness in the system. 2) It also is responsible for setting the rescheduling flag (`TIF_NEED_RESCHED`) if there is more than one task on that run queue, which is used by the scheduler to schedule out the task if the reschedule flag is set. We implemented the rescheduling strategy by bypassing the setting up of the reschedule flag in case the `preempt_notifier` check function returned true, meanwhile updating the runtime statistics of the vCPU.

Annotating locks for *e*`CS`. We mark *e*`CS` by using the non-preemptable methods for non-blocking primitives, preemptable ones for `mutex` and `rwsem`. Our annotation comprises only 60 LoC that covers around 12,000 lock instances with 85,000 lock method calls in the Linux kernel that has 10 million LoC for the kernel version 4.13.

4.4 Evaluation

We evaluate our approaches by answering the following questions:

- What is the overhead of an *e*`CS` annotation and the scheduler overhead to read the values? (§4.4.1)
- Does *e*`CS` helps in an over-committed case? (§4.4.2)
- How does *e*`CS` impact the scalability of a VM? (§4.4.3)
- How do our methods address the `BWW` problem? (§4.4.4)
- Does our schedule penalization approach maintain the eventual fairness of the system? (§4.4.5)

Experimental setup. We extended `VBench` [126] for our evaluation. We chose four benchmarks: Apache web server [127], Metis [128], Psearchy from Mosbench, and Pbzip2 [129]. The Apache web server serves a 300 bytes static page for each request that is generated by WRK [130]. Both of them are running inside the VM to remove the network wire overhead and only stress the VM's kernel components. We choose Apache to stress the interrupt handler to emphasize the importance of *e*`CS` for an interrupt context. Metis is a map-reduce library for a single multi-core server that mostly stresses the memory allocator (spinlock) and the page-fault handler (`rwsem`) of the OS. Similar to Metis, Psearchy is an in-memory parallel search and indexer that stresses the writer side of the `rwsem` design. In addition, we also choose Pbzip2—a parallel compression and decompression program—because we wanted to use a minimally kernel-intensive application. Moreover, none of these workloads suffer from performance degradation from any known user space bottleneck in a non-virtualized environment. We use memory-based file system, `tmpfs`, to isolate the effect

Table 4.4: Cost of using our lightweight para-virtualized methods with various synchronization primitives and mechanism. *1 core* and *80 core* denote the time (in ns) to execute an empty critical section with one and 80 threads, respectively. Although, our approach slightly adds an overhead on a single core count, there is no performance degradation for our evaluated workloads.

Critical sections	Time (ns)			
	1 core		80 core	
	W/o eCS	W/ eCS	W/o eCS	W/ eCS
API cost	–	16.4	–	16.4
spinlock	31.2	44.8	4,782.3	4,772.9
rwlock (read)	32.0	38.8	2,418.2	2,519.4
rwlock (write)	27.4	45.8	4,363.3	4,784.5
mutex	33.5	34.4	49,116.4	48,125.7
rwsem (read)	35.6	36.6	2,588.8	2,737.0
rwsem (write)	33.3	38.1	7,055.7	7,150.1
RCU	9.8	19.7	9.8	19.8

of I/O. We further pin the cores to circumvent vCPU migration at the hypervisor level to remove the jitter from our evaluation.

We evaluate our eCS approach against the following configurations: 1) PVM is a para-virtualized VM that includes unfair qspinlock implementation, which mitigates LWP and BWW issues, which already incorporates the idea of oticket as well as preemptable ticket locks [131]. This is the default configuration since Linux v4.5. 2) HVM is the one without para-virtualization support and also includes unfair qspinlock implementation. Both PVM and HVM are not eCS annotated. Note that we could not compare other prior works because they are not open sourced [121, 116] and are very specific to the Xen hypervisor [86]. We evaluate these configuration on an eight socket, 80-core machine with Intel E7-8870 processors. Another point is that the current version of KVM partially addresses the BWW problem that can occur from the user space [132].

4.4.1 Overhead of eCS

We evaluate the cost of our lightweight para-virtualized methods on various blocking and non-blocking locks, and RCU. Table 4.4 enumerates the overhead of the sole method cost including the cost of executing a critical section with a simple microbenchmark that executes

an empty critical section to quantify the impact of eCS method on these primitives in both lowest (1 core) and highest contention (80 core) scenarios. *1 core* denotes that a thread is trying to acquire a critical section, whereas *80 core* denotes that 80 threads are competing. We observe that eCS adds an overhead of almost 0.9–18.4 ns in low contention, whereas negligible overhead in high contention scenario, except RCU. For RCU, the empty critical section suffers from almost twice the overhead because both RCU's lock/unlock operations do a single memory update on the preempt_count variable for a preemptable kernel. Even though our methods add an overhead in the low contended scenario, we do not observe any performance degradation for any of our evaluated workloads.

4.4.2 Performance in an Over-committed Scenario

We evaluate the performance of the aforementioned workloads in an over-committed scenario by running two VMs in which each vCPU from both VMs share a physical CPU. Figure 4.2 and Figure 4.3 (i) show the throughput of these workloads for PVM, HVM, and eCS; (ii) show the number of unavoidable preemptions that we capture while running these workloads when a vCPU is about to be scheduled out for eCS; and (iii) represent the percentage of types of observed preemptions, namely, LHP for blocking (B-LHP) and non-blocking (NB-LHP) locks, RP, RRP, ICP problems that we observe for the eCS configuration, including both avoided and unavoided preemptions.

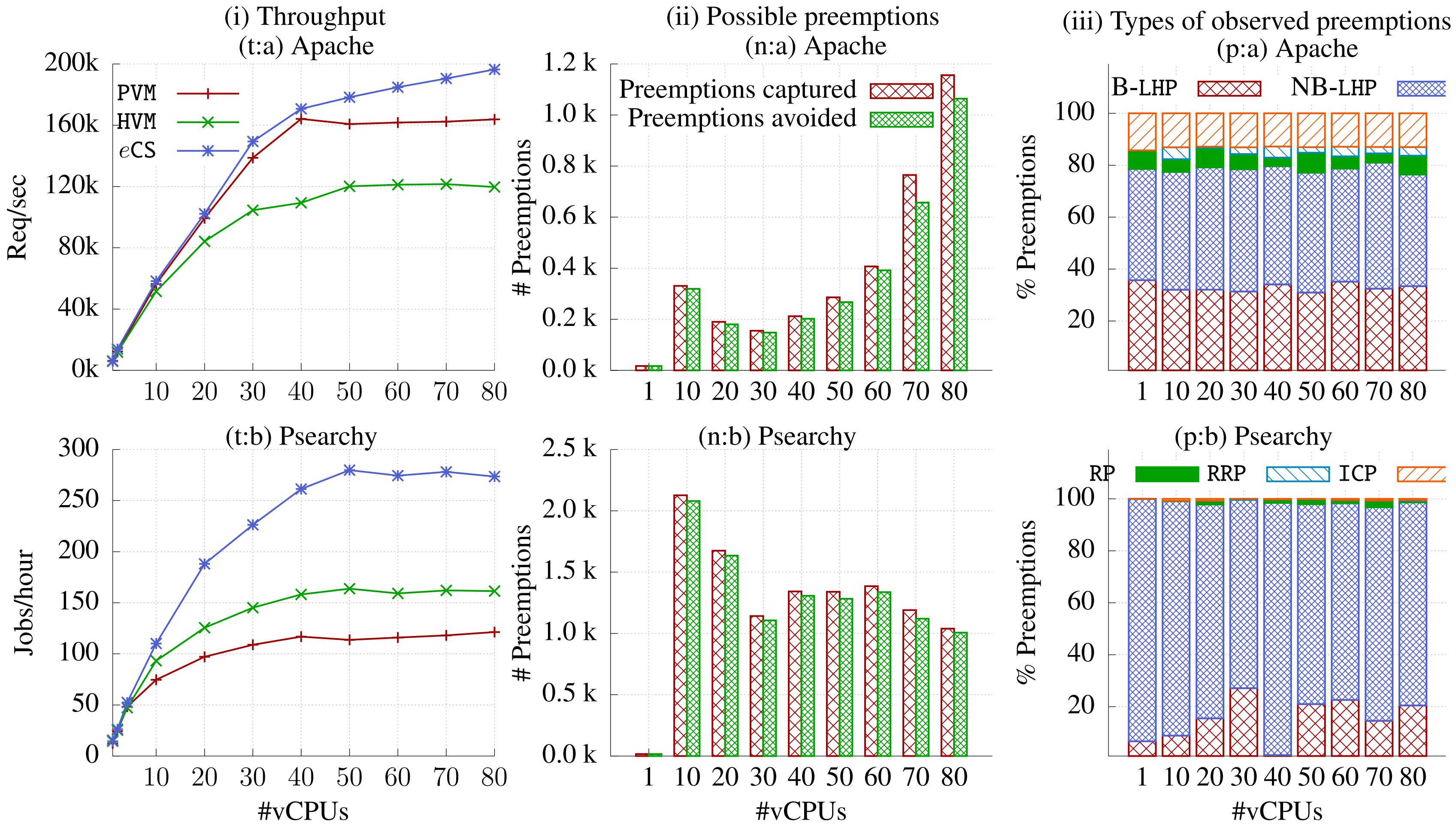

Figure 4.2: Analysis of real-world workloads (Apache and Psearchy) in an over-committed scenario, *i.e.*, two instances of VM are executing the same workload. Column (i) represents the scalability of selected workloads in three settings: PVM, HVM, and with *e*CS annotations. Column (ii) represents the number of preemptions caught and prevented by the hypervisor with our methods. Column (iii) represents the type of preemptions caught by the hypervisor (refer Table 4.4). By allowing an extra schedule, our approach reduces preemptions by 85–100% and improve scalability of applications by up to 2.5×, while observing almost all types of preemptions for each workload.

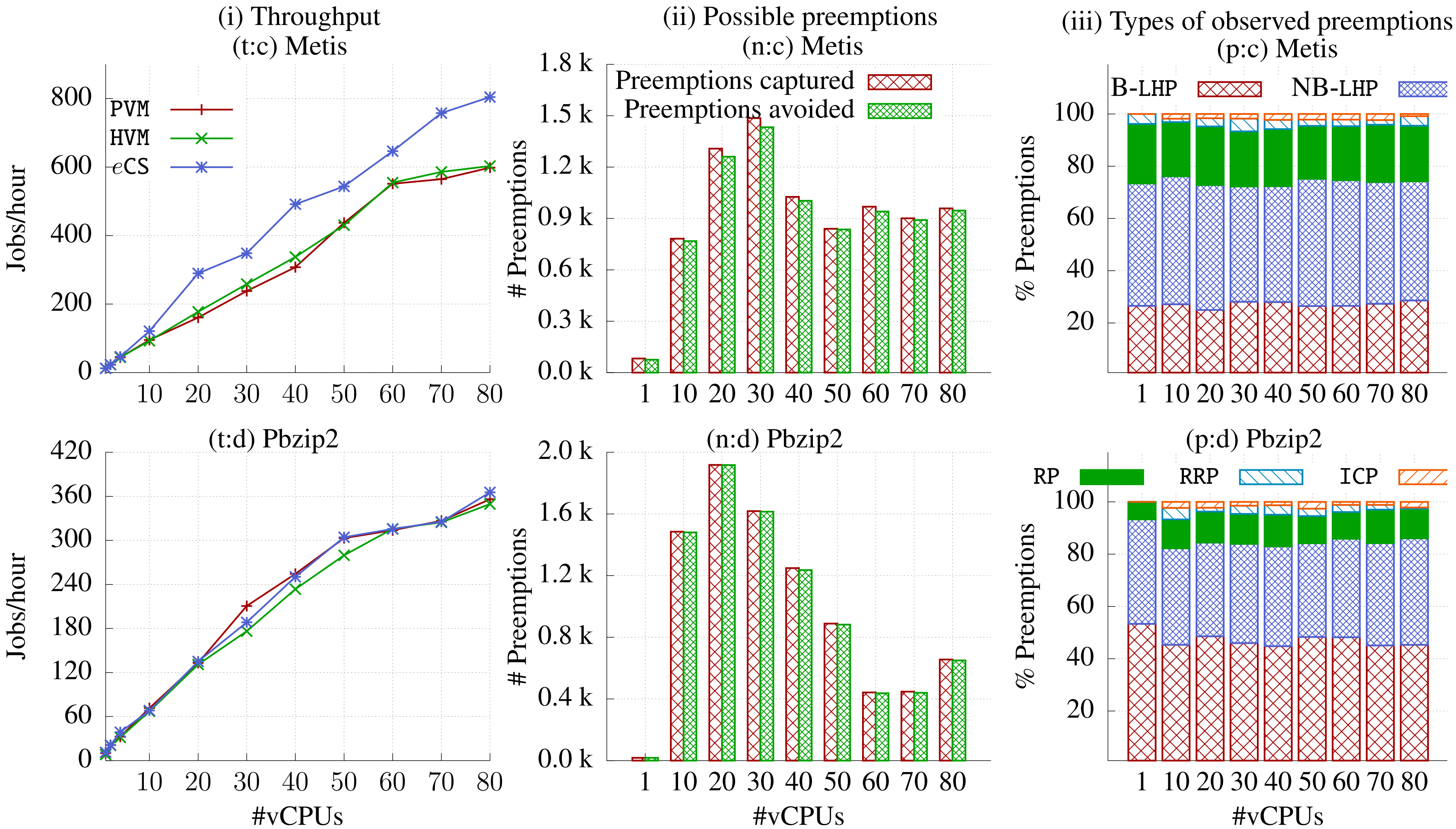

Figure 4.3: Analysis of real-world workloads (Metis and Pbzip2) in an over-committed scenario, *i.e.*, two instances of VM are executing the same workload. Column (i) represents the scalability of selected workloads in three settings: `PVM`, `HVM`, and with *e*`CS` annotations. Column (ii) represents the number of preemptions caught and prevented by the hypervisor with our methods. Column (iii) represents the type of preemptions caught by the hypervisor (refer Table 4.4). By allowing an extra schedule, our approach reduces preemptions by 85–100% and improve scalability of applications by up to 2.5×, while observing almost all types of preemptions for each workload.

Apache. *e*CS outperforms both PVM and HVM by 1.2× and 1.6×, respectively (refer (t:a) in Figure 4.2). Moreover, our approach reduces the number of possible preemptions by 85.8–100% (refer (n:a)) because of our rescheduling approach. We cannot completely avoid all preemptions because of our schedule penalization approach, as some of the preemptions occur consecutively. Even though *e*CS adds overhead, especially to RCU, it still does not degrade the scalability for four reasons: 1) We address the BWW problem, which allows for more opportunities to acquire the lock on time; 2) both hypervisor → VM methods allow cooperative co-scheduling of the VMs; 3) our extra schedule approach avoids 85.8–100% of captured preemptions with the help of our VM → hypervisor methods; and 4) the methods overhead partially mitigates the highly contended system at higher core count by acting as a back-off mechanism. Another interesting observation is that we observe almost every type of preemption (refer Figure 4.2 (p:a)) because of serving the static pages, which involves blocking locks for the socket connection and softirq and spinlocks use for the interrupts processing. In particular, the number of preemptions is dominated by LHP for non-blocking and blocking locks, followed by ICP and then RP. We believe that the ICP problem will further exacerbate with optimized interrupt delivery mechanisms [93, 94]. PVM is 1.36× faster than HVM at 80 cores because of the support of para-virtualized spinlock (qspinlock [87]) as well as the asynchronous page fault mechanism that decreases the contention [133].

The major bottleneck for this workload is the interrupt injection, which can be mitigated by proposed optimized methods [93, 94]. In addition, Figure 4.4 (b) presents the latency CDF for the Apache workload at 80 cores in both under- and over-compression case. We observe that *e*CS not only maintains almost equivalent latency as that of PVM in an under-committed case, but also decreases in the over-committed case by 10.3–17% and 9.5–27.9% against PVM and HVM, respectively.

Psearchy mostly stresses the writer side of rwsem as it performs 20,000 small and large mmap/munmap operations along with stressing the memory allocator for inode operations, which mostly idles the guest OS because of the non-work conserving blocking locks [66].

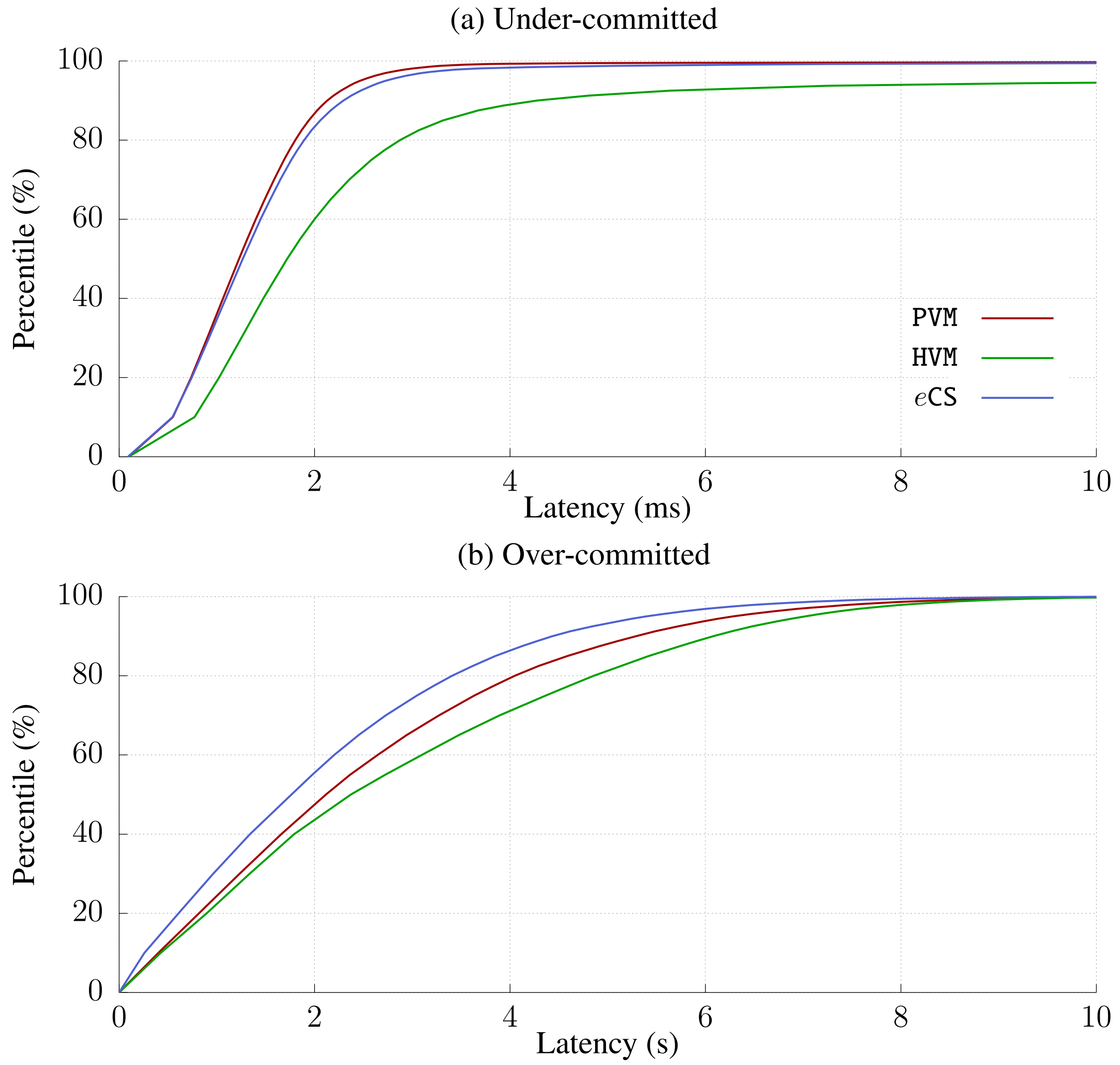

Figure 4.4: CDF of the latency of requests for the Apache web server workload in both under- and over-committed scenarios at 80 cores. It clearly shows the impact of eCS in the over-committed scenario, while having minimal impact in the under-committed case.

Figure 4.2 (t:b) shows the throughput, in which eCS outperforms both PVM and HVM by 2.3× and 1.7×, respectively. The reason is that we 1) partially mitigate the BWW problem with our eSCHDSPIN approach, and 2) decrease the number of preemptions by 95.7–100% with an extra schedule (refer (n:b)). In addition, our eSCHDSPIN approach decreases the idle time from 65.4% to 45.2%, as it allows waiters to spin than schedule out themselves, which severely degrades the scalability in a virtualized environment, as observed for both PVM

and HVM. This workload is dominated by mostly blocking and non-blocking locks, as they account to almost 98% preemptions (refer (p:b)). We also observe that HVM outperforms PVM by 1.33× because the asynchronous page fault mechanism introduces more BWW issue as it schedules out a vCPU if the page is not available, which does not happen for HVM.

Metis is a mix of both page fault and mmap operations that stress both the reader and the writer of the rwsem. Hence, it also suffers from the BWW problem, as we observe in Figure 4.3 (t:c). *e*CS outperforms PVM and HVM by 1.3× at 80 cores because of the reduced BWW problem and decreased preemptions that account to 91.4–99.5% (Figure 4.3 (n:c)). Note that the reader preemptions are 20%, thereby illustrating that readers preemptions is possible for read-dominated workloads, which has not been observed by any prior works. We do not observe any difference in the throughput of HVM and PVM.

Pbzip2 is an efficient compression/decompression workload that spends only around 5% of the time in the kernel space. Figure 4.3 (t:d) shows that the performance of *e*CS is similar to PVM and HVM, while decreasing the number of preemptions by 98.4–100% (refer (n:d)). We do not observe any performance gain in this scenario because 1) these preemptions may not be too critical to affect the application scalability, and 2) the overhead of our methods, which do not provide any gains even after decreasing the preemptions. Similar to the other workloads, LHP dominates the preemption, followed by RP, ICP, and RRP.

In summary, our methods not only reduce preemptions by 85–100%, but also improve the scalability of applications that use these synchronization primitives up to 2.5×, while no observable overhead on these applications. Moreover, we found that these preemptions occur for almost every type of primitives, specifically in the case of blocking synchronization primitives, read locks (Metis and Pbzip2), and interrupts (*e.g.*, TLB operations, packet processing etc.). In addition, most of the workloads still suffer from the BWW problem because of them being non-work conserving. We partially address this problem with the help of our *e*SCHDSPIN approach. One point to note is that we do not observe too many preemptions, as shown by prior works [86], because the current Linux kernel has dropped the

FIFO-based Ticket spinlock and has replaced it with a highly optimized unfair queue-based lock [87] that mitigates the problem of LHP and LWP.

4.4.3 Performance in an Under-committed Case

We evaluate our *e*CS approach against PVM and HVM configurations in which a VM is running to show the impact of both methods and *e*SCHDSPIN approach. We also include bare-metal configuration (Host) as a baseline (Figure 4.5). We observe that *e*CS addresses the BWW problem, and outperforms both PVM and HVM in the case of Apache (1.2× and 1.2×), Psearchy (1.6× and 1.9×), Metis (1.2× and 1.3×), and Psearchy (1.2× and 1.4×), while having almost similar latency for the Apache workload (Figure 4.4 (a)). Likewise, *e*CS performance is similar to that of bare-metal, except for the Psearchy workload.

For Apache, our methods act as a back-off mechanism to improve its scalability, as the system is heavily contended. The throughput degrades after 30 cores because of the overhead of process scheduling, socket overhead, and inefficient kernel packet processing. Besides this, both Psearchy and Metis suffer from the BWW problem, which we improve with our *e*SCHDSPIN approach that results in better scalability as well as reduction in the idling of VMs. In particular, we decrease the idle time of Psearchy and Metis by 25% and 20%, respectively, by using our approach. One point to note is that blocking locks are based on the TAS lock, whose throughput severely degrades with increasing core count because of the increase cache-line contention, which we observe after 40 cores for Psearchy for all configurations. We also find that the Host is still 1.4× faster than *e*CS because *e*SCHDSPIN only partially mitigates the BWW problem, while introducing excessive cache-line contention, which we can circumvent with NUMA-aware locks [66]. For Pbzip2, we observe that *e*CS performs equivalent to the Host, while outperforming PVM and HVM after 60 cores, because Pbzip2 spends the least amount of time in the kernel space (5%), and starts to suffer from the BWW problem only after 60 cores, which our *e*SCHDSPIN easily tackles.

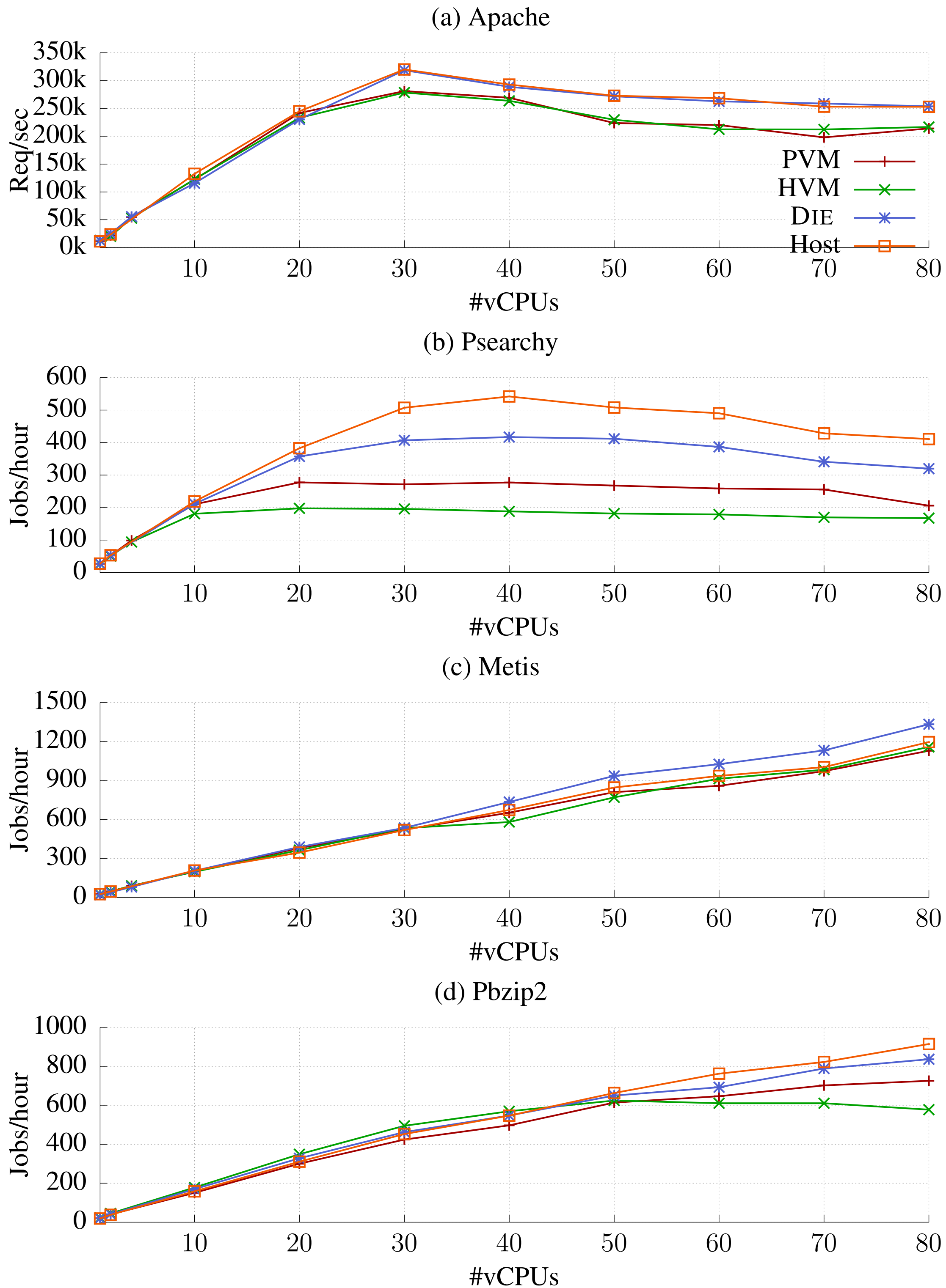

Figure 4.5: Performance of real-world workloads when running on the bare metal (Host), and inside a VM with three configurations: `PVM`, `HVM`, and with *e*`CS` annotations. In this scenario, only one VM is running. We use Host as the baseline for the comparison because we consider Host to have almost optimal performance.

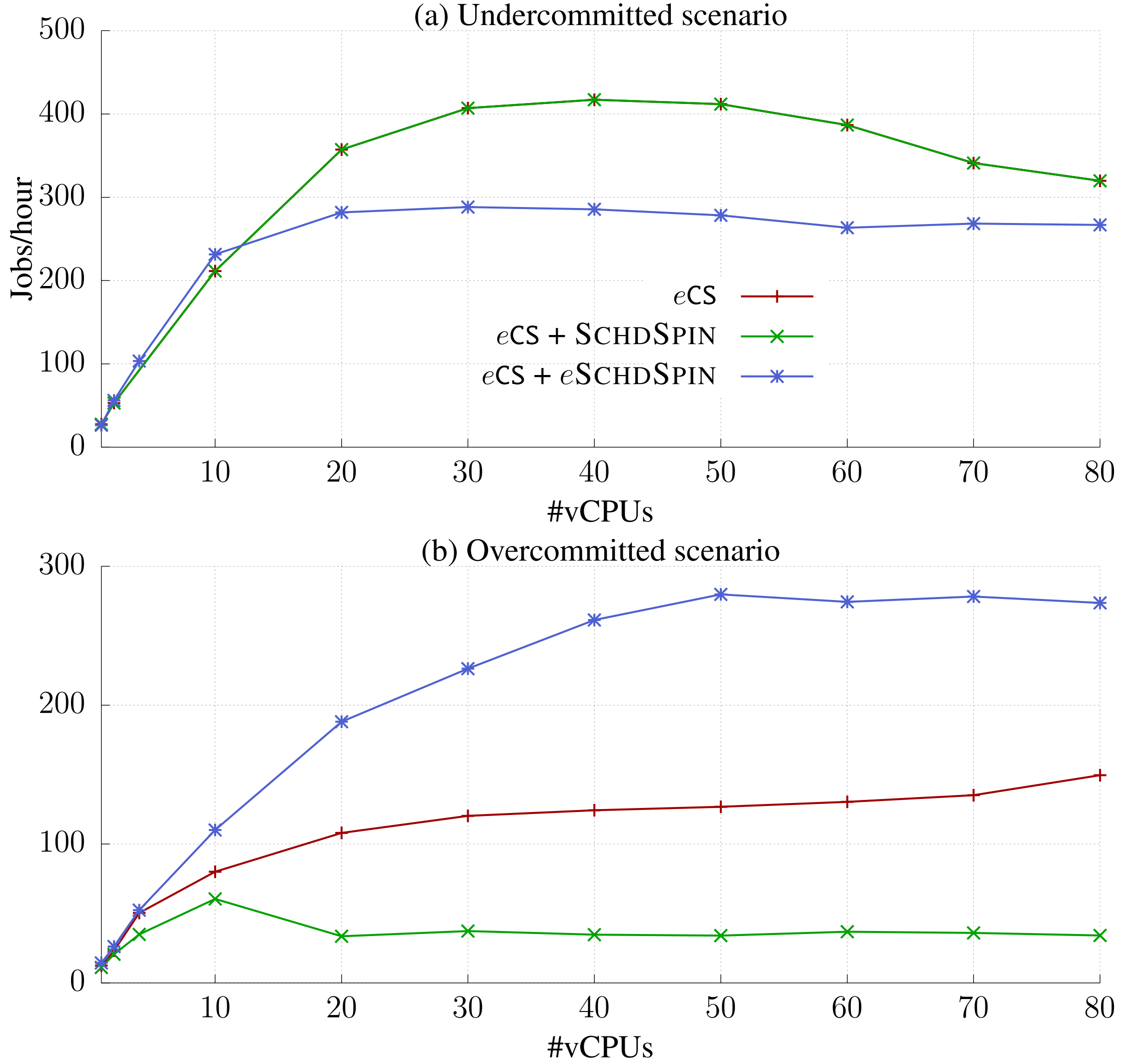

Figure 4.6: Impact of both BWW problem and *e*CS method (refer Hypersivor → VM in Table 4.1) on Psearchy in both under- and over-committed scenarios.

4.4.4 Addressing BWW Problem via *e*CS

We evaluate the impact of the BWW problem on Psearchy in both under- and over-committed scenarios. Figure 4.6 (a) shows that our scheduling-aware spinning approach (marked as *e*CS + SCHDSPIN) improves the throughput of Psearchy by 1.5× and 1.2× at 40 and 80 cores, respectively, in an under-committed scenario. SCHDSPIN approach allows a blocking waiter, both reader and writer, to actively spin for the lock if the number of tasks

in the run queue is one, else the task schedules itself out. This approach is similar to the scheduling-aware parking/wake-up strategy [66], which we applied to the stock `mutex` and `rwsem`. As mentioned before, the reason for such an improvement is that the current design is not scheduling aware, as the waiter parks itself if it is unable to acquire the lock. With our approach, we try to mitigate this performance anomaly and allow the applications to scale further. Unfortunately, the scheduling-aware approach is inefficient in the case of the over-committed scenario, as shown in Figure 4.6 (b). The reason is that current waiters are guest OS agnostic, which leads to wasting CPU resources and resulting in more LHP and LWP problems, thereby degrading the scalability by almost 4.4× (marked *e*CS + SCHDSPIN in (b)) against a simple *e*CS configuration that still suffers from the BWW problem. We overcome this issue by using our `is_pcpu_overcommitted()` method that allows the SCHDSPIN approach to spin only when there is no active task on the pCPU's run queue; otherwise, the waiter is scheduled out when more than one task are in the run queue of the pCPU. By using our method (marked *e*CS + *e*SCHDSPIN), we outperform the baseline *e*CS approach by 1.8× and the *e*CS + SCHDSPIN approach by 8×.[2]

4.4.5 System Eventual Fairness

We now evaluate whether we are able to achieve eventual fairness while allowing *e*CS annotated VMs to obtain an extra schedule followed by local vCPU penalization. To evaluate the fairness, we run a simple micro-benchmark in two VMs (marked VM_1 and VM_2). VM_1 is a non-annotated VM, whereas VM_2 is an *e*CS annotated one. This micro-benchmark indefinitely reads the content of a file that stresses the read side of the `rwsem` and spends around 99% of the time in the kernel without scheduling out the task, thereby prohibiting the guest OS from doing any halt exits. Figure 4.7 (a) shows the time difference between two VM runtimes that we measure at every 100 ms window for each VM as well as the number of preemptions for VM_2 in that window. Figure 4.7 (b) shows the cumulative runtime of the VMs. We observe

[2]We have used *e*CS + *e*SCHDSPIN approach for our evaluation against PVM and HVM in §4.4.2 and §4.4.3.

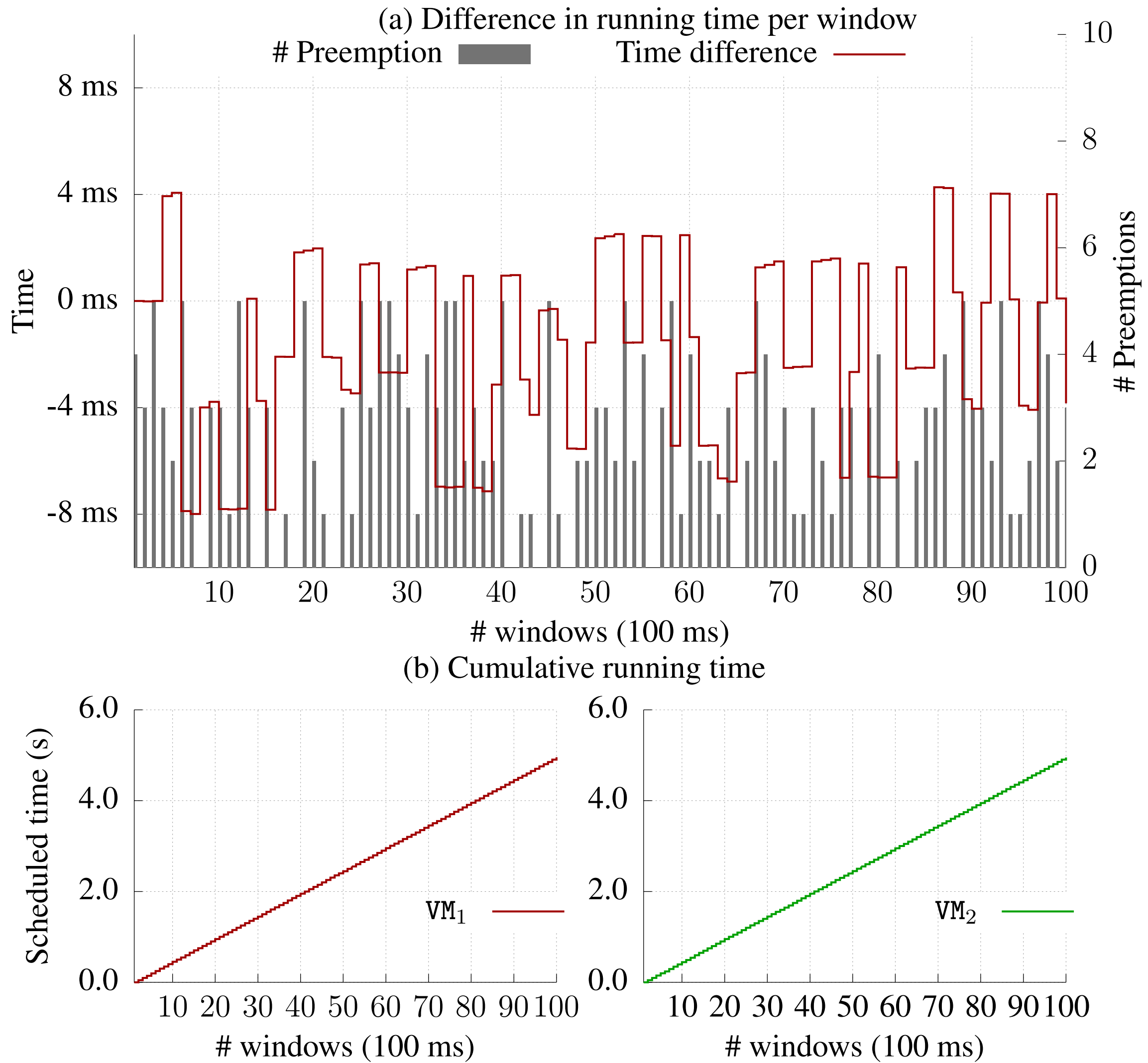

Figure 4.7: Fairness in *e*CS. Running time of a vCPU of two co-scheduled VMs ($\mathtt{VM}_1$ and $\mathtt{VM}_2$) with *e*CS annotations for a period of 10 seconds with 100 ms window granularity while executing a kernel intensive task (reading the contents of a file) that involves read side of `rwsem`. (a) shows the difference in running time of vCPU per window granularity as well as the number of preemptions occurring per window, while (b) illustrates the cumulative running time, and shows that the hypervisor maintains eventual fairness in the system, even if $\mathtt{VM}_2$ is allowed extra schedules. Both VMs get 4.95 seconds to run.

from Figure 4.7 (a) that even after allowing for extra schedules, the CFS scheduling policy balances out these extra schedules, which does not affect the runtime difference between $\mathtt{VM}_1$ and $\mathtt{VM}_2$. For example, at the end of one second window, marked 10, we observe that the

number of extra schedules that the hypervisor granted $\texttt{VM}_2$ was 34 (34 milliseconds of extra time), but the runtime difference between $\texttt{VM}_1$ and $\texttt{VM}_2$ is 7.8 ms, which becomes -1.9 ms at the end of two seconds, while $\texttt{VM}_2$ received a total of 54 extra schedules (54 milliseconds). Hence, the extra schedule approach followed by our local vCPU penalization ensures that none of the tasks running on that particular physical CPU suffers from the fairness issue, also referred as eventual fairness. Moreover, Figure 4.7 (b) shows that both VMs get almost equivalent runtime in a lockstep fashion with both VMs getting almost 4.95 seconds at the end of 10 seconds.

4.5 Chapter Summary

Double scheduling phenomenon is a well-known problem in the domain of virtualization that leads to several symptoms in the form of LHP, LWP, and BWW. We identify that it not only is limited to non-blocking locks, but also is applicable to blocking locks and reader side of locks. We present a single shot solution with our key insight: if a certain key component of a guest OS is allowed to proceed further, the guest OS will make forward progress. We identify these critical components as synchronization primitives and mechanism such as spinlocks, mutex, rwsem, RCU, and even interrupt context, which we call *enlightened critical sections* (*e*CS). We annotate *e*CS with our lightweight methods that expose whether a VM is executing a critical section, which the hypervisor uses to provide an extra schedule at the scheduling boundary, thereby allowing the guest OS to progress forward. In addition, by leveraging the hypervisor scheduling context, a VM mitigates the effect of BWW problem with our simple virtualized spinning-aware spinning strategy. With *e*CS, we not only decrease the spurious preemptions by 85–100% but also improve the throughput of applications up to 1.6× and 2.5× in an under- and over-committed scenario, respectively.

CHAPTER 5

SCALABLE LOCKING PRIMITIVES

Since the invention of concurrent programming, lock design has been influenced by hardware evolution. For instance, MCS [59] was proposed to address excessive cache-line traffic resulting from an increasing number of threads trying to acquire the lock at the same time, while Cohort locks [63] were proposed in response to the emergence of the non-uniform memory access (NUMA) architecture. NUMA machines consist of multiple nodes (or sockets), each with multiple cores, locally attached memories, and fast caches. In such machines, the access from a socket to its local memory is faster than remote access to memory on a different socket [62] and each socket has a shared last-level-cache. Cohort locks exploit this characteristic to improve application throughput.

Unfortunately, The influence of hardware evolution on lock design has resulted in a tight coupling between hardware characteristics and lock algorithms. Meanwhile, other factors have been neglected, such as memory footprint [134], low thread counts, and core over-subscription.

This chapter investigates the main dominating factors that impact the scalability of locks and their adoption: 1) cache-line movement between different caches, 2) level of thread contention, 3) core over-subscription, and 4) memory footprint. We propose a new lock design technique called **shuffling** that decouples the lock-acquire/release phases from the lock order policy, and uses lock waiters (*i.e.*, threads waiting to acquire the lock) to enforce those policies, mostly off the critical path. In shuffling, a waiter in the waiting queue takes the role of a shuffler and re-orders the queue of waiters based on the specified policy. This technique gives us the freedom to design and multiplex new policies based not only on the characteristics of fast-evolving hardware, but also on software characteristics. Our new family of locks, called SHFLLOCKS, augment existing locks (TAS and MCS) and use

shuffling. Our first lock algorithm is a non-blocking lock that implements NUMA-awareness as a shuffling policy to implement a compact NUMA-aware lock. We further add a core over-subscription policy to implement a blocking lock. We also implement a readers-writer lock on top of our blocking lock. We evaluate our locks in both kernel space and in userspace, and find that our lock algorithms maintain the best throughput regardless of the number of threads contending for the lock. In particular, SHFLLOCKS improve application throughput by 1.2–12.5×, while reducing the memory footprint up to 35.4% and 98.8%, against the currently used Linux kernel locks and against state-of-the-art locks, respectively.

5.1 Dominating Factors in Lock Design

Locks not only serialize data access, but also add their overhead, directly impacting application scalability. Looking at the evolution of locks and their use, we identify four main factors that any practical lock algorithm should consider. These factors are critical in achieving good performance in current architectures, but their relative importance can vary not only across architectures, but also across applications with varying requirements. Therefore, we should holistically consider all four factors when designing a lock algorithm. Table 5.1 shows how these factors impact state-of-the-art locks.

F_1. Avoid data movement. Memory bandwidth and the interconnect bandwidth between NUMA sockets are limited, leading to performance bottlenecks when applications incur remote cache traffic or remote memory accesses. Thus, every lock algorithm should minimize cache-line movement and remote memory accesses for both lock structures and data inside the critical section. This movement is quite expensive in NUMA machines: the cost of accessing a remote cache line can be 3× higher than local access [32]. Moreover, for future architectures, even L1/L2 cache-line movements will further exacerbate this cost [135]. Similarly, for readers-writer locks, their readers indicator incurs cache-line movement. *A lock algorithm should amortize data movement from both the lock structure and the data inside the critical section, to hide non-uniform latency and minimize coherence traffic.*

Table 5.1: Dominant factors affecting locks that are in use in the Linux kernel or are the state-of-the-art for NUMA architecture. Cache-line movement refers to shfllock/data movement inside a critical section. Boxes represent the scalability of locks with increasing thread count from one thread to threads within a socket to all threads among multiple sockets. Core subscription is only applicable to blocking locks and denotes the best throughput for a varying number of threads. Both `mutex` and `CST` are sub-optimal when under-subscribed but maintain good throughput once they are over-subscribed. Memory footprint is the memory allocation for locks: the size of each lock instance (per lock), a queue node required by each waiting thread before entering the critical section (per waiter), and a queue node retained by a lock holder within the critical section (per lock-holder). If the lock holder uses the queue node, which happens for MCS, CNA, and Cohort locks, the thread must keep track of the node, as it can acquire multiple locks: a common scenario in Linux. Note that queue nodes can be allocated on the stack for each algorithm. However, in practice, a lock user needs to explicitly allocate it on the stack for MCS, CNA, and Cohort locks, while `mutex`, `CST`, and SHFLLOCKS avoid this complexity. We also summarize the number of atomic instructions in the non-contended/contended scenarios.

	Factors		Non-blocking locks				Blocking locks		DIE
			TAS	**MCS** [59]	**Cohort** [63]	**CNA** [136]	**mutex** [122]	**CST** [66]	
F_1.	*Cache-line movement*		Very high	High	Low	Low	High	Low	Low
F_2.	*Thread contention*								
F_3.	*Core subscription (U/O)*		-	–	–	–	×/ ✓	×/ ✓	✓/ ✓
F_4.	*Memory footprint (bytes)*	Per lock:	1	8	1,152‡	8	40	1,056‡	12
		Per waiter:	0	12	24	28	32	24	28
		Per lock-holder:	0	12	24	28	0	0	0
Atomic instructions per acquire and release:			$1/\infty$	2/1	$4/\approx 2$	$2/\approx 2$	$1/\approx 4$	$\approx 6/\approx 3$	$1/\approx 2$

U: Under-subscribed; O: Over-subscribed. ‡For CST and Cohort locks, we use an eight-socket machine as a reference.

■ → 1–2 threads ■ → 1 socket ■ → > 1 socket ■ → Optimal throughput ■ → Sub-optimal throughput ■ → Worst throughput

F_2. Adapt to different levels of thread contention. Most multi-threaded applications use fine-grained locking to improve scalability. For example, Dedup and fluidanimate [137] create 266K and 500K locks, respectively. Similarly, Linux has also adopted fine-grained locking over time (Figure 2.2) and only a subset of locks heavily contend based on the workload [46]. Generally, lock designs optimize either for low contention or for high contention: TAS results in better performance when contention is low, while Cohort locks are a better choice for high contention. Similarly, the scalability of a readers-writer lock is determined by its low-level design and choices, such as using a centralized readers indicator vs. per-socket indicators vs. per-core indicators impact scalability depending on the ratio of readers and writers. *For the best performance in all scenarios, a lock algorithm should adapt to varying thread contention.*

F_3. Adapt to over- or under-subscribed cores. Applications can instantiate more threads than available cores to parallelize tasks, to improve hardware utilization, or to efficiently deal with I/O. In these scenarios, **blocking locks** need to efficiently choose between spinning or sleeping, based on the thread scheduling. Spinning results in the lowest latency, but can waste CPU cycles and underutilize resources while starving other threads, leading to lock-holder preemption [138]. In contrast, sleeping enables threads to run and utilize the hardware resources more efficiently. However, this can result in latency as high as 10ms to wake up a sleeping thread. *Thus, a lock algorithm should consider the mapping between threads and cores and whether cores are over-subscribed.*

F_4. Decrease memory footprint. The memory footprint of a lock not only affects its adoption, but also indirectly affects application scalability. Generally, the structures of a lock are not allocated inside the critical section or on the critical path, so many algorithms do not consider these allocations as a performance overhead. However, in practical applications, locks are embedded inside other structures, which can be instantiated on the critical path. In such scenarios, this allocation aggravates the memory footprint, which stresses the memory allocator, leading to performance degradation. For example, Exim, a mail server, creates

three files for each message it delivers. Locks are part of the file structure (inode), so large locks can slow down allocation and directly affect performance [134]. This is even worse for locks that dynamically allocate their structure before entering the critical section [66]. The memory allocation can fail, leading to an application crash. Extra per-task or per-CPU allocations can further exacerbate the issue, *e.g.*, for queue-based locks [139, 140]. Memory footprint also affects readers-writer scalability because the memory consumption dramatically increases for the readers indicators from centralized (8 bytes) to per-socket (1 KB) to per-CPU (24KB) for each lock instance.[1] *Thus, a lock algorithm should consider memory footprint, as it affects both the adoption of the lock and applications performance.*

5.2 SHFLLOCKS

To adapt to such a diverse set of factors, we propose a new lock design technique, called *shuffling*. Shuffling enables the decoupling of lock operations from a lock policy enforcement, which happens off the critical path. Policies can include NUMA-awareness and efficient parking/wakeup strategies. Using shuffling, we design and implement a family of lock algorithms called SHFLLOCKS. At its core, a SHFLLOCK uses a combination of TAS as a top-level lock and a queue of waiters (similar to MCS). We rely on the shuffling mechanism to enable NUMA-awareness that minimizes cache-line movement (F_1). SHFLLOCKS work well under high contention due to their NUMA-awareness, while maintaining good performance for low contention due to their TAS lock (F_2). Besides NUMA-awareness, we also add a parking/wakeup policy to design an efficient blocking SHFLLOCK (F_3). SHFLLOCKS requires a constant, minimal data structure and does not require additional allocations within the critical section, thereby reducing memory footprint (F_4).

[1] Per-socket: 8 sockets × 128 bytes; per-CPU: 192 cores × 128 bytes.

5.2.1 The Shuffling Mechanism

Shuffling is a new technique for designing locks in which a thread waiting for the lock (the shuffler) re-orders the queue of waiters (shuffles) based on a policy specified by the lock developer. Shuffling is similar to sorting a list with a user-defined comparison function. Here, the list represents a set of waiters and the comparison function is a set of policies, such as NUMA-awareness or a wakeup/parking strategy. This shuffling mechanism is mostly off the critical path because a thread handles the task of policy enforcement while waiting to acquire the lock. Thus, shuffling enables the decoupling of lock-acquire/release operations from policy enforcement, and allows lock developers to easily optimize for particular design factors (§5.1) or architectures. In this paper, we use a policy designed to optimize for NUMA architectures. Moreover, shuffling can group multiple policies together to devise complex lock algorithms. For example, in the blocking SHFLLOCK we combine the NUMA-aware policy with an efficient parking/wakeup strategy: the shuffler groups waiters based on their NUMA socket and wakes up a nearby sleeping waiter. This approach solves the lock-waiter preemption problem by removing the wake-up overhead from the lock-holder critical path, a well-known issue for queue-based locks [66, 141, 142].

5.2.2 SHFLLOCKS Design

We now present a family of SHFLLOCK protocols, both non-blocking (§5.2.2) and blocking (§5.2.2). We further augment our blocking lock with a read indicator to design a blocking readers-writer lock (§5.2.2). We first enumerate a set of design decisions and later focus on the design of these locks.

Lock state decoupling. Unlike the MCS protocol, we decouple the lock acquisition state from the waiter queue. We achieve decoupling by introducing two levels of locks: a `TAS` lock for handling non-contended cases and a queue-based lock to handle moderate contention at the socket level. This approach is similar to the Linux spinlock and has several foundational benefits for building practical and scalable locks: a) SHFLLOCKS remove the complexity of

node allocation and tracking for the waiters queue because a queue node is only maintained within the acquire phase. This contrasts with conventional MCS/CNA locks, which maintain the node until the release phase. This prevents the lock-holder from reusing the node for a nested acquisition; b) SHFLLOCKS use waiters for shuffling, moving work from the critical path to threads that are waiting; c) SHFLLOCKS provide a fast trylock method with a single atomic compare-and-swap instruction; and d) SHFLLOCKS mitigate the lock-waiter preemption problem through two mechanisms. First, the shuffler wakes up the next thread to acquire the lock proactively (§5.2.1). Second, SHFLLOCKS allow lock stealing using the internal TAS lock.

Scheduling-aware parking strategy. For a blocking synchronization primitive, the most important question is how to efficiently pass the lock or wake up a waiter, while maintaining an on-par performance in both the under- and over-subscribed cases. For the scalable parking/wake-up decision, we remove costly scheduler operations (i.e., wake-up) from the common, critical path and employ a distributed parking decision while considering the load on a system. We discuss three key ideas to address the problem of 1) whom to pass the lock to, 2) when to park oneself, and 3) how to take the parking decisions for blocking synchronization primitives.

➊ **Passing lock to an active spinning waiter.** In queue-based locks (e.g., MCS, K42, and CLH), the successor of a lock holder always acquires the lock, which guarantees complete fairness, but, unfortunately, causes severe performance degradation in an over-subscribed system, as this invariant stresses the scheduler to always issue a call to wake up the parked waiter. To mitigate this issue, we introduce lock stealing, in which waiters that are about to join the queue, can steal the lock in the absence of the lock holder. In addition, the shuffler also tries to wake up sleeping threads to keep a set of actively spinning waiters.

➋ **Scheduling-aware spinning.** Most of the proposed hierarchical locks [63, 68, 64] are non-blocking in nature. Hence, they do not consider the amount of time a waiter should spin before parking itself out. Thus, in an over-loaded system, waiting threads and a lock holder

will contend with each other, which deters the system progress. Instead, in SHFLLOCK, waiting threads park themselves as soon as their time quota is about to cease. To check the quota, we rely on the scheduler and its APIs for this information. Specifically in the Linux kernel, the scheduler exposes `need_resched()` to know whether the task should run, and preemption APIs (`preempt_disable() / preempt_enable()`) to explicitly disable or enable the task preemption. These APIs work with both preemptive and non-preemptive kernels. Limiting the duration of spinning up to the time quota proposed by the scheduler has several advantages: 1) It guarantees the forward progress of the system in an over-loaded system by allowing the current lock holder to do useful work while mitigating its preemption. 2) It allows other tasks to do some useful work rather than wasting the CPU cycles. 3) By only spinning for the specified duration, the primitive respects the fair scheduling decision of the scheduler.

❸ **Scheduling-aware parking.** The current blocking synchronization primitives [89, 90] do not efficiently account for the system load; thus, they naively park waiters even in under-loaded scenarios. Hence, a naive use of the spin-then-park approach results in scheduler intervention, as the waiters park themselves as soon as their time quota ceases, and the lock holder has to do an extra operation of waking them up, which severely degrades the performance of the system in an under-loaded scenario [77]. Also, previous research [143] has shown that estimating system load is critical to the spin-then-park approach because it not only removes the scheduler interaction from the parking phase, but also improves the latency of the lock/unlock phase.

We gauge the system load by peeking at the number of running tasks on a CPU (i.e., the length of scheduling run queue for a CPU). Checking the number of running tasks is almost free because a modern OS kernel, including Linux, has a per-CPU scheduler queue, which already maintains an up-to-date per-CPU active task information. On the other hand, maintaining system-wide, central information, like the approach used by Johnson et al. [143], is costly because the cost of collecting the total number of active tasks increases

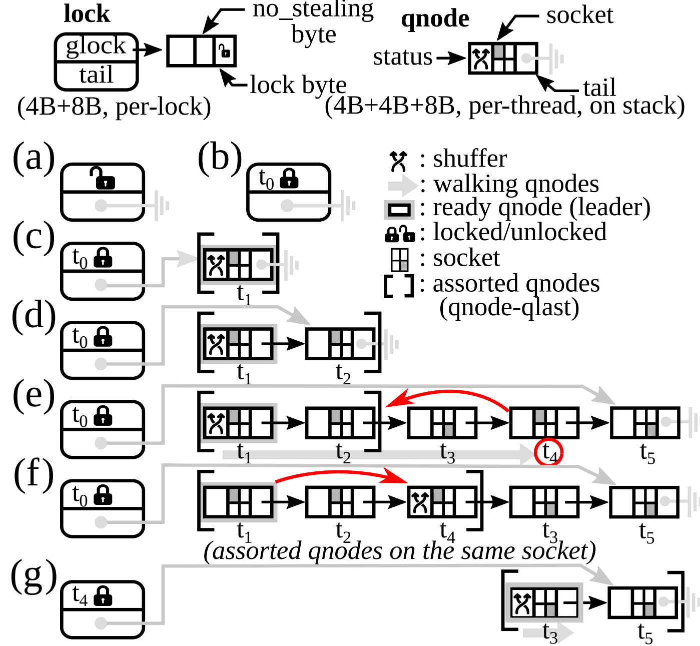

Figure 5.1: SHFLLOCK^{NB} example. The lock structure consists of a state (`glock`) and the queue tail. The first byte of `glock` is the lock/unlock state, while the second byte denotes whether stealing is allowed. We encode multiple information in the qnode structure. (a) Initially, there is no lock holder. (b) t_0 successfully acquires the lock via `CAS` and enters the critical section. (c) t_1, of socket 1, executes `SWAP` on the lock's tail after the `CAS` failure on `TAS`. (d) Similarly, t_2 from socket 1, also joins the queue. (e) Now, there are five waiters (t_1–t_5) waiting for the lock. t_1 is the very first waiter, so it becomes the shuffler and traverses the queue to find waiters from the same socket. t_1 then moves t_4 (same socket) after t_2. (f) After the traversal, t_1 selects t_4 as the next shuffler. (g) t_4 acquires the lock after t_1 and t_2 have executed their critical sections. At this point, t_3 becomes the shuffler.

with increasing core count, which may not catch the load imbalance due to the new incoming tasks or the rescheduling of periodic tasks.

Non-Blocking Version: SHFLLOCK^{NB}

SHFLLOCK^{NB} uses a `TAS` and MCS combination, and maintains queue nodes on the stack [140, 139, 66]. However, we do extra bookkeeping for the shuffling process by extending the thread's qnode structure with socket ID, shuffler status, and batch count (to limit batching too many waiters from the same socket, which might cause starvation or break long-term fairness). Figure 5.1 shows the lock structure and the qnode structure. Our current design of the shuffling phase enforces the following four invariants for implementing any policy: 1) The successor of the lock holder, if it exists, always keeps its position intact in the

queue. 2) Only one waiter can be an active shuffler, as shuffling is single threaded. 3) Only the head of the queue can start the shuffling process. 4) A shuffler may pass the shuffling role to one of its successors.

Figure 5.1 presents a running example of our lock algorithm. (a) A thread first tries to acquire the `TAS` lock; (b) it enters the critical section on success; otherwise, it joins the waiting queue ((c)–(e)). Now, the very next lock waiter, *i.e.*, $\texttt{t}_1$, becomes the shuffler and groups waiters belonging to the same socket, *e.g.*, $\texttt{t}_4$ (Figure 5.1 (e)). Once a shuffler iterates the whole waiting queue, it selects the last moved waiter as the next shuffler to start the process: $\texttt{t}_1$ selects $\texttt{t}_4$ (f). The shuffler keeps retrying to find a waiter from the same socket and leaves the shuffling phase after finding a successor from the local socket (f) or becoming the lock holder (g). The passing of a shuffler status, within a socket, lasts until the batching quota is exceeded.

Listing 5.1 presents the pseudo-code of our non-blocking version. The lock structure is 12 bytes (Figure 5.1): 4 bytes for the lock state (`glock`), and 8 bytes for the MCS tail. The algorithm works as follows: A thread `T` first tries to steal the `TAS` lock (line 6). On failure, `T` initiates the MCS protocol by first initializing a queue-node (`qnode`) on the stack, and then adding itself to the waiting queue by atomically swapping the tail with the `qnode`'s address (line 11–13). After joining the queue, `T` waits until it is at the head of the queue. To do that, `T` checks for its predecessor. If `T` is the first one in the queue, it disables lock stealing by setting the second byte to 1 to avoid `TAS` lock contention and waiter starvation (line 17). On the other hand, if waiters are present, `T` starts to spin locally until it becomes the leader in the waiting queue, *i.e.*, until its `qnode`'s status changes from `S_WAITING` to `S_READY` (line 47). Here, `T` also checks for the `is_shuffler` status. If the value is set, then `T` becomes the shuffler and enters the shuffling phase (line 51), which we explain later.

On reaching the head of the queue, `T` checks whether it can be a shuffler to group its successors based on the socket ID, meanwhile trying to acquire the `TAS` lock via the `CAS`

```
S_WAITING = 0 # Waiting on the node status
S_READY   = 1 # The waiter is at the head of the queue

def spin_lock(lock):
  # Try to steal/acquire the lock if there is no lock holder
  if lock.glock == UNLOCK && CAS(&lock.glock, UNLOCK, LOCKED):
    return

  # Did not get the node, time to join the queue; initialize node states
  qnode = init_qnode(status=S_WAITING, batch=0,
                     is_shuffler=False, next=None, skt=numa_id())

  qprev = SWAP(&lock.tail, &qnode) # Atomically adding to the queue tail
  if qprev is not None:  # There are waiters ahead
    spin_until_very_next_waiter(lock, qprev, &qnode)
  else: # Disable stealing to maintain the FIFO property
    SWAP(&lock.no_stealing, True) # no_stealing is the second byte of glock

  # qnode is at the head of the queue; time to get the TAS lock
  while True:
    # Only the very first qnode of the queue becomes the shuffler (line 16)
    # or the one whose socket ID is different from the predecessor
    if qnode.batch == 0 or qnode.is_shuffler:
      shuffle_waiters(lock, &qnode, True)
    # Wait until the lock holder exits the critical section
    while lock.glock_first_byte == LOCKED:
      continue
    # Try to atomically get the lock
    if CAS(&lock.glock_first_byte, UNLOCK, LOCKED):
      break

  # MCS unlock phase is moved here
  qnext = qnode.next
  if qnext is None: # qnode is the last one / next pointer is being updated
    if CAS(&lock.tail, &qnode, None): # Last one in the queue, reset the tail
      CAS(&lock.no_stealing, True, False) # Try resetting, else someone joined
      return
    while qnode.next is None: # Failed on the CAS, wait for the next waiter
      continue
    qnext = qnode.next
  # Notify the very next waiter
  qnext.status = S_READY

def spin_until_very_next_waiter(lock, qprev, qcurr):
  qprev.next = qcurr
  while True:
    if qcurr.status == S_READY:  # Be ready to hold the lock
      return
    # One of the previous shufflers assigned qcurr as a shuffler
    if qcurr.is_shuffler:
        shuffle_waiters(lock, qcurr, False)

def spin_unlock(lock):
  lock.glock_first_byte = UNLOCK # no_stealing is not overwritten
```

```
55 MAX_SHUFFLES = 1024
56
57 # A shuffler traverses the queue of waiters (single threaded)
58 # and shuffles the queue by bringing the same socket qnodes together
59 def shuffle_waiters(lock, qnode, vnext_waiter):
60   qlast = qnode # Keeps track of shuffled nodes
61   # Used for queue traversal
62   qprev = qnode
63   qcurr = qnext = None
64   # batch → batching within a socket
65   batch = qnode.batch
66   if batch == 0:
67     qnode.batch = ++batch
68   # Shuffler is decided at the end, so clear the value
69   qnode.is_shuffler = False
70   # No more batching to avoid starvation
71   if batch >= MAX_SHUFFLES:
72     return
73
74   while True: # Walking the linked list in sequence
75     qcurr = qprev.next
76     if qcurr is None:
77       break
78     if qcurr == lock.tail:  # Do not shuffle if at the end
79       break
80
81     # NUMA-awareness policy: Group by socket ID
82     if qcurr.skt == qnode.skt: # Found one waiting on the same socket
83       if qprev.skt == qnode.skt: # No shuffling required
84         qcurr.batch = ++batch
85         qlast = qprev = qcurr
86
87       else: # Other socket waiters exist between qcurr and qlast
88         qnext = qcurr.next
89         if qnext is None:
90           break
91         # Move qcurr after qlast and point qprev.next to qnext
92         qcurr.batch = ++batch
93         qprev.next = qnext
94         qcurr.next = qlast.next
95         qlast.next = qcurr
96         qlast = qcurr  # Update qlast to point to qcurr now
97     else: # Move on to the next qnode
98       qprev = qcurr
99
100    # Exit → 1) If the very next waiter can acquire the lock
101    #         2) A waiter is at the head of the waiting queue
102    if (vnext_waiter is True and lock.glock_first_byte == UNLOCK) or
103           (vnext_waiter is False and qnode.status == S_READY):
104      break
105
106  qlast.is_shuffler = True
```

Listing 5.1: Pseudo-code of the non-blocking version of SHFLLOCKS and the shuffling mechanism.

operation (lines 20–30). Note that only the head of the queue can start the shuffling process if the qnode's batch is set to 0. Otherwise, T can only shuffle waiters if the is_shuffler status is set to 1, which might be set by a previous shuffler.

The moment T becomes the lock holder, *i.e.*, T acquires the TAS lock, it follows the MCS unlock protocol (lines 33–40). T checks for the next successor (qnode.next). If the successor is present, T updates the successor's qnode status to S_READY. Otherwise, it tries to reset the queue's tail and enables lock stealing, which enables a new thread to get the lock via TAS if the queue is empty. The unlock phase is a conventional TAS unlock in which the first byte is reset to 0 (line 54).

Shuffling. Our shuffling algorithm moves a waiter's qnode from an arbitrary position to the end of the shuffled nodes in the waiting queue. Based on the specified policy, *i.e.*, socket-ID-based grouping, the shuffler (S) either updates the batch count or further manipulates the next pointer of waiting qnodes (line 82–98). We consider S as the first shuffled node. The algorithm is as follows: S first resets its is_shuffler to 0 and checks its quota of the maximum allowed shufflings to avoid starvation for remote socket waiters (line 69–71). Similar to CNA, we can also use a random generator to mitigate starvation. Now, S iterates over qnodes in the queue while keeping track of the last shuffled qnode (qlast). While traversing, S always marks the nodes that belong to its socket by increasing the batch count. It only does pointer manipulations when there are waiters between the last shuffled node and the node belonging to S's socket (lines 87–96). Finally, S always exits the shuffling phase if either the TAS lock is unlocked or S becomes the head of the queue (line 102–103). Before exiting the shuffling phase, S assigns the next shuffler: the last marked node (line 106). S can stop traversing the queue for two more reasons: 1) if successors are absent (line 76, 89), as S wants to avoid the locking delay because it might soon acquire the lock; 2) if S reaches the queue tail, as there might be waiters joining at the end of the tail, which it cannot move (line 78).

Optimization. Our shuffling algorithm has unnecessary pointer chasing when a newly

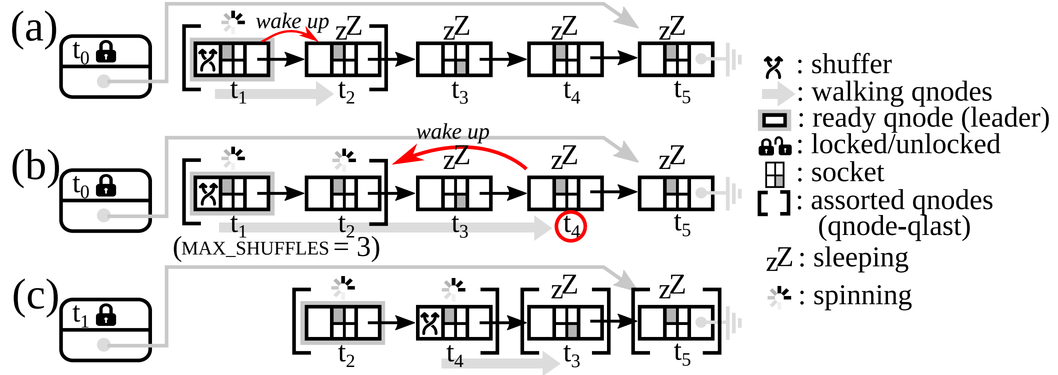

Figure 5.2: A running example of how a shuffler shuffles waiters with the same socket ID and wakes them up. (a) t_0 is the lock holder; t_1 is the shuffler and is traversing the queue. t_2 is sleeping, but t_1 wakes it up. (b) t_2 becomes active, while t_1 continues shuffling and reaches t_4, t_1 first moves t_4 after t_2, and wakes up t_4 to mitigate the wakeup latency. (c) When t_0 releases the lock, t_1 acquires it; t_2 and t_4 are actively spinning for their turn; t_4 is the shuffler.

selected shuffler, assigned by the previous `S`, has to traverse the queue. We avoid this issue by further encoding extra information about the qnode where `S` stopped traversal in the next shuffler's qnode structure. This leads to traversing mostly from the near end of the tail, thereby better utilizing the time of waiters.

Blocking Version: SHFLLOCK^B

We augment SHFLLOCK^{NB} to incorporate an effective parking/wakeup policy. Our lock algorithm departs from the scalable queue-based blocking designs as we do not have a separate parking list [66, 122, 144]. This allows us to save up to 16–20 bytes per lock compared to existing separate parking list-based locks. We maintain both the active and passive waiters in the same queue, and utilize the `TAS` lock for lock stealing and shuffling to efficiently wake up parked waiters off the critical path. SHFLLOCK^B avoids the lock-waiter preemption by allowing the `TAS` lock to be unfair in the fast path [66, 140] as well as keeping the head of the waiting queue active, *i.e.*, not scheduled out. In addition, we modify the MCS protocol to support waiter parking and wakeup. We further extend our shuffling protocol to wake up the nearby sleeping waiters while shuffling the queue for NUMA-awareness in

```
1  + S_PARKED   = 2 # Parked state (used by lock waiter for sleeping)
2  + S_SPINNING = 3 # Spinning state (used by shuffler for waking up)
3    def mutex_lock(lock):
4    ...
5      # Notify the very next waiter
6  -   qnext.status = S_READY
7  +   # Atomically SWAP the qnode status
8  +   prev_status = SWAP(&qnext.status, S_READY)
9  +   if node_pstate == S_PARKED: # Required for avoiding lost wakeup
10 +     wake_up_task(qnext.task) # Explicitly wake up the very next waiter
11
12   def spin_until_very_next_waiter(lock, qprev, qcurr):
13   ...
14       if curr.status == S_READY:
15         return
16 +     if task_timed_out(qcurr.task): # Running quota is up! Give up
17 +       park_waiter(qcurr) # Will try to park myself
18
19   def shuffle_waiters(lock, qnode, next_flag):
20   ...
21     if batch >= MAX_SHUFFLINGS:
22       return
23 +   SWAP(&qnode.status, S_SPINNING) # Don't sleep, will soon acquire the lock
24
25     while True:
26   ...
27       # NUMA-awareness and wakeup policy
28       if qcurr.skt == qnode.skt:
29         if qprev.skt == qnode.skt: # No shuffling required
30 +         update_node_state(qcurr) # Disable sleeping
31           qnode.batch = ++batch
32           qlast = qcurr
33           qprev = qcurr
34
35         else:
36 +         update_node_state(qcurr) # Disable sleeping
37           qnode.batch = ++batch
38           qprev.next = qnext
39
40 + def update_node_state(qnode):
41 +   # If the task is waiting, then make it spinning
42 +   if CAS(&qnode.status, S_WAITING, S_SPINNING):
43 +     return
44 +   # If the task is sleeping, then wake it up for spinning
45 +   if CAS(&qnode.status, S_PARKED, S_SPINNING):
46 +     wake_up_task(qnode.task) # Wakeup task (off the critical path)
47 +
48 + def park_waiter(qnode):
49 +   # Park it when the task is waiting
50 +   if CAS(&qnode.status, S_WAITING, S_PARKED):
51 +     park_task(qnode.task)
```

Listing 5.2: The extra modification required to convert our non-blocking version of SHFLLOCK to a blocking one.

```
1   def mutex_lock(lock):
2   ...
3 +   qnext = qnode.next  # Try to get the successor before acquiring TAS
4 +   if qnext is not None:
5 +     if SWAP(&qnext.status, S_SPINING) == S_PARKED:
6 +       wake_up_task(qnext.task)
7
8     # qnode is at the head of the queue; time to get the TAS lock
9     while True:
10  ...
```

Listing 5.3: An optimization for avoiding a waiter wakeup issue in the critical path with an extra state update before the `TAS` lock.

both under- and over-subscribed cases (Figure 5.2). To support efficient parking/wakeup, we extend our non-blocking version with two more states: 1) *parked* (`S_PARKED`), in which a waiter is scheduled out for handling core over-subscription and 2) *spinning* (`S_SPINNING`), in which a shuffled waiter is always spinning for mitigating the convoy effect.

Listing 5.2 shows the modifications on top of SHFLLOCK^{NB}. While spinning locally on its status, a waiter `T` checks if the time quota is up (line 16). In that case, `T` tries to atomically change its qnode status from `S_WAITING` to `S_PARKED` (line 50). On success, `T` parks itself out (line 51); otherwise, `T` goes back to spinning. In the shuffling phase, a shuffler `S` also wakes up the shuffled sleeping waiters (lines 30, 36). Note that this is a best effort strategy, in which an `S` first tries to atomically `CAS` the `qnode`'s status from `S_WAITING` to `S_SPINNING`, hoping that the waiter is still waiting locally; if the operation fails, then `S` does another explicit `CAS` from `S_PARKED` to `S_SPINNING` and wakes up the sleeping waiter if successful (line 46). The last notable change to the algorithm is notifying the head of the queue. There is a possibility that the very next waiter might be sleeping. We atomically swap the `qnext`'s state to `S_READY` (line 8) and wake up the waiter at the head of the queue if the return value of the atomic `SWAP` operation is `S_PARKED` (line 10).

Optimizations. Our first optimization is to enable lock stealing by not setting the second byte when the queue begins. The reason is that waking up a waiter ranges from 1μs–10ms, which adds overhead in the acquire phase. The second optimization regards the waiter

wakeup. Our current design leads to waking up the queue head inside the critical section, even though it is rare (see §4.4). As shown in Listing 5.3, we explicitly set the successor status to `S_SPINNING` and wake it up if parked. This approach further removes the rare occurrence of the waiter preemption problem at the cost of an extra atomic operation, which is acceptable, as the atomic operation is only between two qnodes. It is not a part of the critical section, as other joining threads can steal the lock (`TAS`) to ensure the forward progress of the system.

Readers-Writer Blocking SHFLLOCK

Linux uses a readers-writer spinlock [145], which combines a readers indicator with a queue-based lock. This lock queues waiting readers and writers to avoid cache-line contention and bouncing. We use a similar design on top of our blocking SHFLLOCK. Thus, our readers-writer lock inherently becomes a blocking lock, and at most only one reader or a writer can spin to acquire, while others spin locally. Our lock design provides only long-term fairness due to the NUMA-awareness of the SHFLLOCK. This is acceptable because even the Linux's `rwsem` is writer-preferred to enhance throughput over fairness [76, 75], similar to prior work [68]. Note that the shuffler only moves writers in the wait queue because all the contiguous readers can enter the critical section together, irrespective of NUMA socket.

Details. We augment a SHFLLOCK, henceforth called `wlock`, with a read/write counter, which encodes: a readers count (`Rcount`), a writer waiting bit (`WWb`) indicating if a writer is waiting to acquire the lock, and a writer byte (`WB`), indicating if a writer is currently holding the lock. A writer enters the critical section on successfully setting `WB` from 0 to 1; otherwise, it enqueues itself to acquire the underlying blocking lock (`wlock`). After acquiring the `wlock`, the writer sets the waiting bit (`WWb`) to 1 to prohibit new readers from entering the critical section and waits for existing readers to leave. Once readers leave, the writer atomically resets `WWb` to 0 and sets `WB` to 1, releases `wlock`, and then enters the critical section. In the writer unlock phase, a writer resets `WB` to 0. A reader first atomically increments `Rcount` and

enters the critical section if both `WB` and `WWb` are 0. If non-zero, the reader first decreases `Rcount` and starts acquiring `wlock`. Once it holds `wlock`, it first increments the `Rcount` to prevent writers from entering the critical section and waits for the existing writer to exit. When `WB` is 0, the reader enters the critical section after releasing `wlock`. In the unlock phase, a reader atomically decreases `Rcount`.

5.3 Implementation

We implement all SHFLLOCKS in the Linux kernel v4.19-rc4 and entirely replace `mutex` and `rwsem` with ours. Our replacement results in adding 459 and 557 lines of code (LoC) for `mutex` and `rwsem`, respectively. We add our shuffling phase to the `qspinlock` in 150 LoC, without increasing the lock size. We have also tested SHFLLOCKS with locktorture.

5.4 Evaluation

We evaluate SHFLLOCKS by answering three questions:

Q1. How do SHFLLOCKS, implemented in the kernel, impact micro-benchmarks (§5.4.1) and real applications (§5.4.2)?

Q2. How does each design decision affect SHFLLOCKS performance and how fair are SHFLLOCKS (§5.4.3)?

Q3. How do userspace SHFLLOCKS impact applications' performance and memory footprint? (§5.4.4)

Evaluation setup. We use micro-benchmarks that stress a single lock [77, 146], and three workloads that heavily stress several kernel subsystems [108, 147]. We also use a hash-table nano-benchmark [148] to break down the performance characteristics of SHFLLOCKS. Table 5.2 lists all the evaluated locks and the selection criteria. We evaluate on an 8-socket, 192-core machine with Intel Xeon E7-8890 v4 (hyperthreading disabled). We use `tmpfs` to minimize file system overhead.

Table 5.2: Locks evaluated in both the kernel space and the userspace. In the kernel space, we replace all locks with SHFLLOCKS. We use LD_PRELOAD to replace all the mutex-based locks in the userspace.

Kernel space		
Locks	**Replaced**	**Selection criteria**
SHFLLOCKS	All	–
CNA [136]	qspinlock	Compact NUMA-aware lock (NB)
CST [66]	mmap_sem / i_rwsem / †	Hierarchical + dynamic allocation (B)
Cohort [63]	s_vfs_rename_mutex †	Hierarchical + static allocation (NB)

Userspace	
Locks	**Selection criteria**
MCS [59]	Queue-based lock (NB)
HMCS [64]	Representative cohort lock (NB)
CNA [136]	Compact version of NUMA-aware MCS (NB)
MCSTP [149]	Preemption adaptive MCS for over-subscription (NB)
pthread	Stock version used by everyone (B)
Mutexee [150]	Optimized version of pthread (B)
Malthusian [144]	Culls extra thread deterministically (B)

B: Blocking; NB: Non-blocking † Both CST and Cohort replace all three locks.

5.4.1 SHFLLOCK Performance Comparison

We evaluate the performance of all SHFLLOCKS using a set of micro-benchmarks [77, 146]. Each micro-benchmark instantiates a set of threads and pins them to cores. These threads contend on a single lock while performing specific tasks (Table 5.3) for 30 seconds. We pin two threads on each core in the over-subscribed scenario for blocking locks.

Non-blocking SHFLLOCK. Figure 5.3 shows that both CNA and SHFLLOCK outperform the Linux version (Stock) by 2.8× and 2× on MWRL and lock1, respectively, while maintaining the same throughput under lower contention, *e.g.*, within a single socket. Similar to SHFLLOCK, CNA maintains NUMA-awareness by using the lock holder to physically split the waiting queue into two, one for local threads and the other for remote threads. Meanwhile, SHFLLOCK uses lock waiters to shuffle waiters around, mostly off the critical path. Like SHFLLOCK, CNA also uses the waiting queue and re-structures this queue to

Table 5.3: Lock usage in various micro-benchmarks [77, 146].

Lock type	Workload	Lock: Usage
Non-blocking	`MWRL` [77] lock1 [146]	**rename seqlock**: Rename files within a directory **files_struct.file_lock**: fd allocation / fcntl
Blocking	`MWRM` [77]	**sb->s_vfs_rename_mutex**: Rename a file across directory
RW blocking	`MWCM` [77] `MRDM` [77]	**inode->i_rwsem**: Create files in the directory (writer) **inode->i_rwsem**: Enumerate files in the directory (readers)

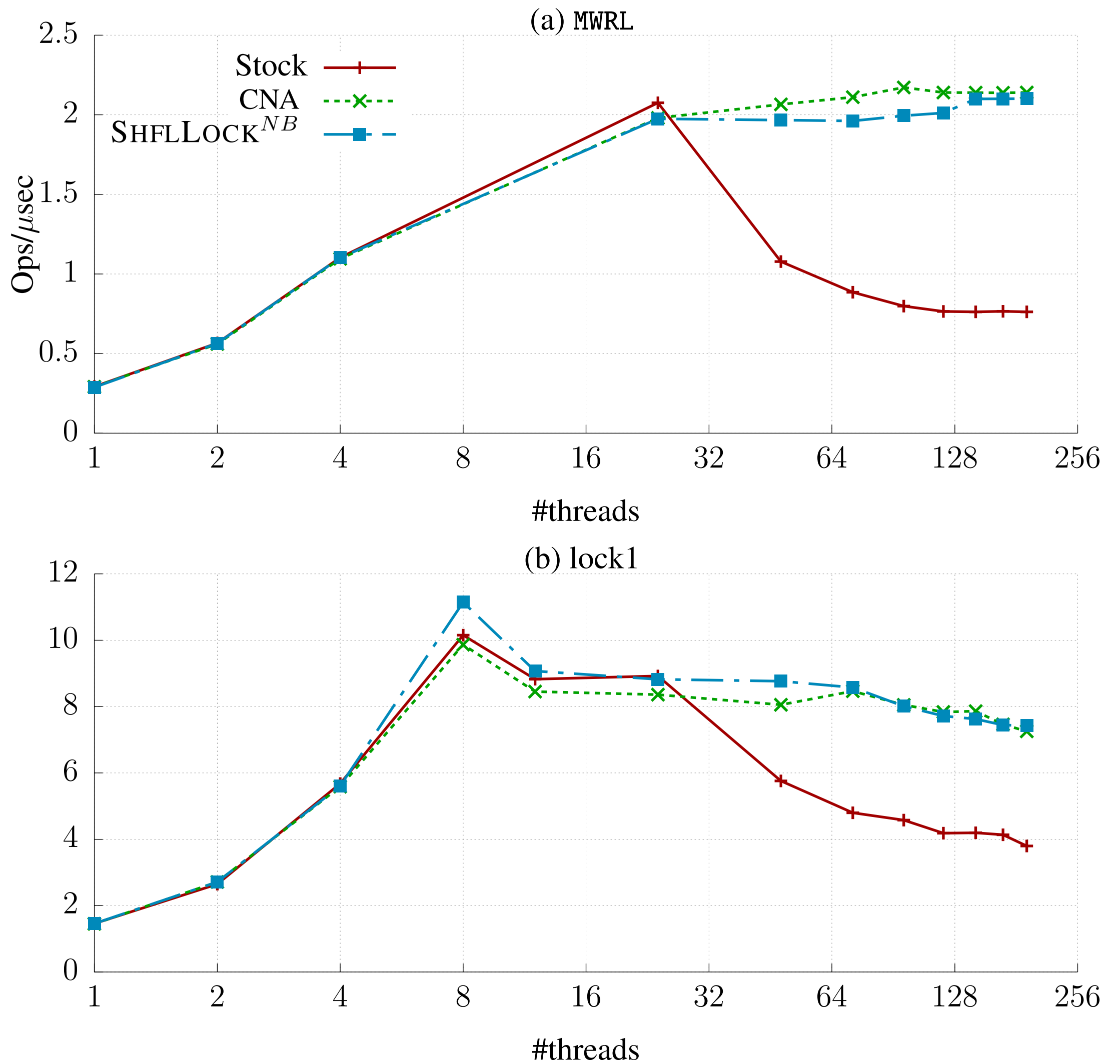

Figure 5.3: Impact of non-blocking locks on the scalability of micro-benchmarks [77, 146]. Refer to Table 5.3 for lock usage. Here, Stock refers to the default spinlock.

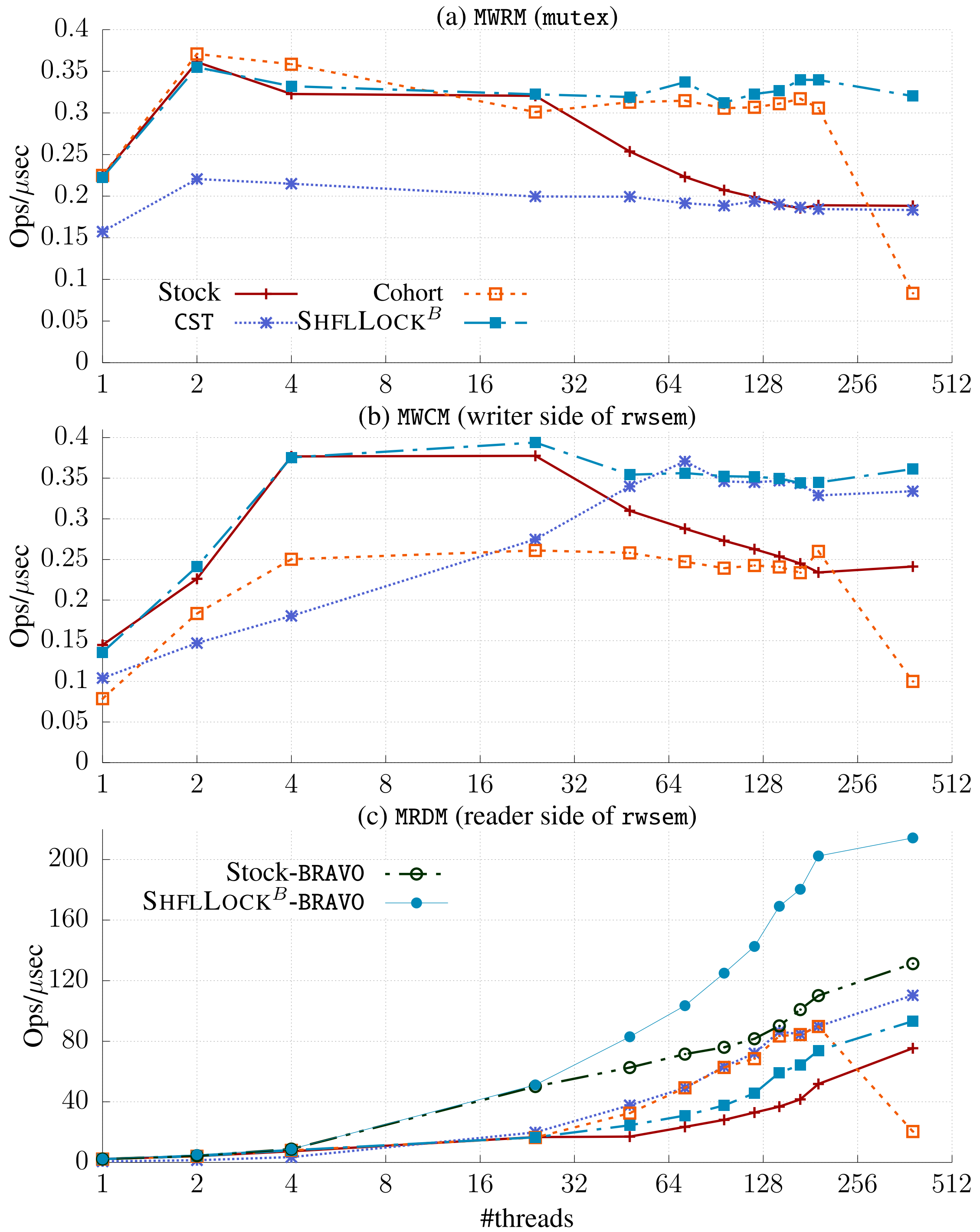

Figure 5.4: Impact of blocking locks on the scalability of micro-benchmarks with up to 2× over-subscription (384 threads: we pin two threads on each core). Cohort and CST are the non-blocking and blocking hierarchical locks, respectively. Refer to Table 5.3 for lock usage.

optimize the next thread to acquire the lock. However, CNA physically partitions this queue into a queue of local NUMA threads and one of remote NUMA threads. This operation results in extending the critical section while acquiring the lock, but at a low amortized cost ($\mathcal{O}(1)$). In contrast, SHFLLOCK generalizes the queue re-structuring by maintaining a single physical queue and using the shuffling operation to prioritize local NUMA threads. Moreover, SHFLLOCK performs this operation *off the critical path*.

Blocking SHFLLOCK. We compare SHFLLOCK with Linux `mutex` and `rwsem` (Stock), Cohort non-blocking lock, and `CST` lock (Table 5.2). We test these locks in both under- and over-subscribed cases, *i.e.*, up to 384 threads by pinning two threads on a core in a round-robin manner. Figure 5.4 (a) shows the results for the `MWRM` benchmark, which renames files across directories. `MWRM` first pre-allocates a set of empty files in per-thread directories; then, each thread moves a file from its directory to a shared directory, which stresses the super-block's mutex (Table 5.3). SHFLLOCK maintains the best throughput in both under- and over-subscribed scenarios and is 1.8× faster than both `CST` and Stock. The stock suffers from cache-line bouncing at high core count but maintains the throughput in the over-subscribed case. Cohort is a non-blocking lock, which performs well up to the total number of cores (192 threads), but significantly degrades `MWRM`'s throughput in the over-subscribed case (384 threads), as waiters waste CPU cycles. `CST` does not scale because it dynamically allocates its socket structure before each critical section, which results in excessive allocations with elongated critical section length. In contrast, Cohort pre-allocates its socket structure, and does not extend the critical section.

Readers-Writer Blocking SHFLLOCK. Figure 5.4 (b) shows the impact of SHFLLOCK when stressing the writer lock of `rwsem`. We use the `MWCM` benchmark, in which each worker creates 4KB files in a shared directory to stress inode allocation. We observe that SHFLLOCK maintains the best throughput at all core counts, due to its ability to better adapt to the workload. For example, SHFLLOCK is 1.8–2× faster than hierarchical locks within a socket and 1.5× faster than Stock at 192 threads. Cohort can only scale up to four cores (almost

55% slower than ShflLock) because memory allocation becomes an issue as the inode size increases by 3.4×. Meanwhile, CST avoids this scenario, as it only allocates the memory for one socket initially, but its performance only scales to reach that of ShflLock after 2 NUMA nodes.

Figure 5.4 (c) shows the impact of ShflLock when stressing the readers side of the rwsem. We use MRDM, in which each thread enumerates files in a directory. We also include a recently proposed approach, called BRAVO [151], that tries to mitigate the centralized reader overhead by using a global readers table. We observe that both hierarchical locks are faster than ShflLock and rwsem because of their per-socket readers indicator, which localizes the contention within a socket. ShflLock is still faster than stock rwsem by 1.2–1.5× because the stock version suffers from the spurious sleeping of waiters, which results in extra cache-line contention on the reader indicator, thereby impacting the throughput. We also observe that the BRAVO approach improves the throughput for both Stock and ShflLock up to 2.3× compared to Cohort and CST locks at 192 threads. However, due to the extra cache-line contention in the stock version, ShflLock-BRAVO still outperforms Stock-BRAVO by 1.6× at 384 threads.

5.4.2 Improving Application Performance

We evaluate three applications that extensively stress various subsystems of the Linux kernel. Figure 5.5 reports the throughput of applications and the memory used by locks, which are mostly blocking and are present in several data structures such as inodes, task structures, and memory mappings. Table 5.2 shows the locks modified for the evaluation. Note that CNA only modifies the spinlock, but does not affect the size of blocking locks.

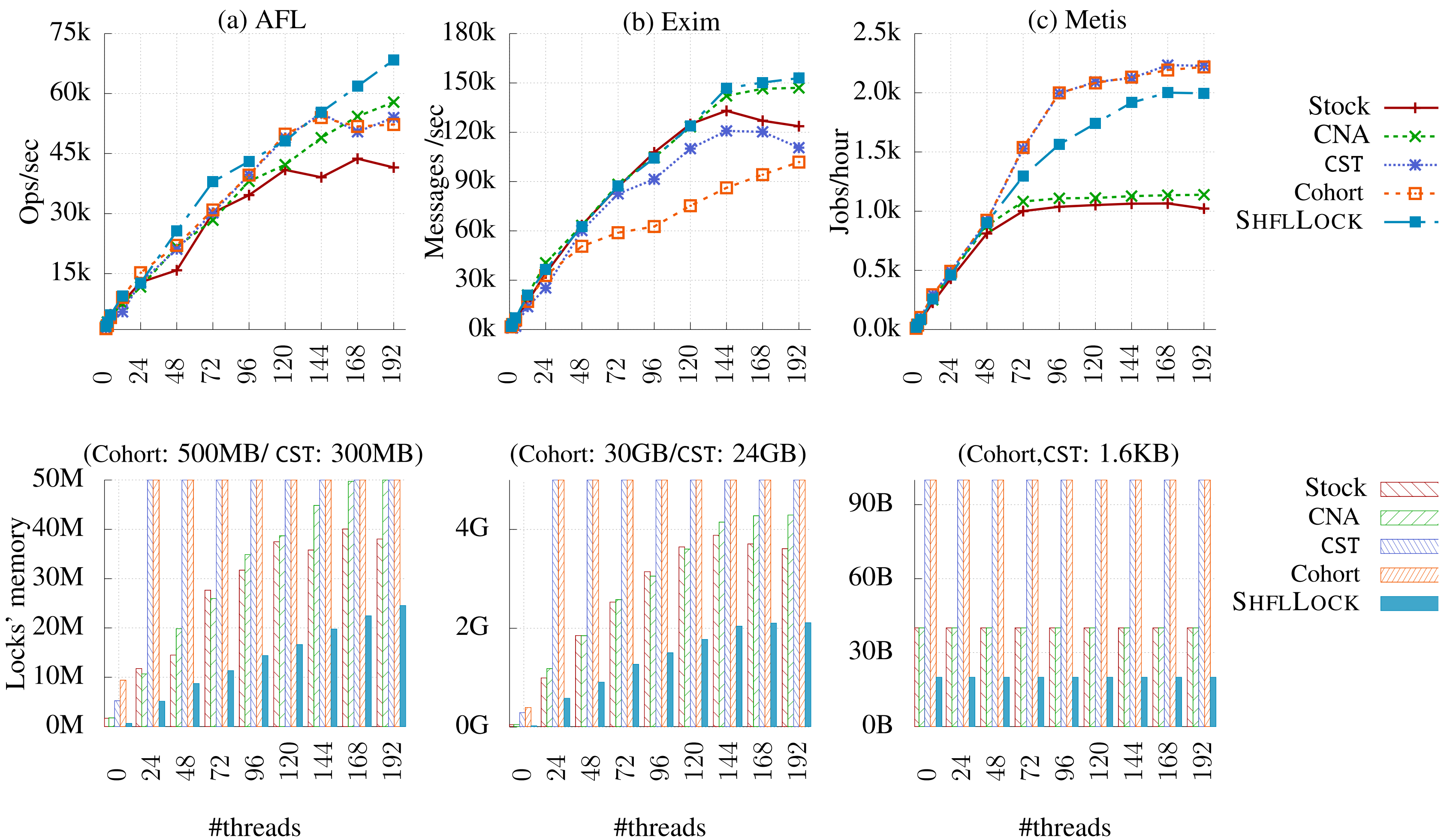

Figure 5.5: Impact of locks on application scalability and on memory footprint, while running three applications with SHFLLOCKS, Linux stock version (Stock), CNA, CST, and Cohort. Refer to Table 5.2 for specific changes. SHFLLOCK reduces the memory footprint because of the blocking locks that are embedded in inodes, task structure, and memory management structures.

AFL [147], a fuzzer, is an embarrassingly parallel workload that heavily uses `fork()` to execute test cases and scan directories created by the fuzzing instances. AFL suffers from the following overheads: process forking, repeatedly creating and unlinking files in a private directory, and scanning other instances' directories [152]. In addition, AFL suffers from the `gettimeofday()` syscall, as each instance issues this syscall to log information. Figure 5.5 (a) shows AFL throughput and memory usage with various locks. We observe that all the existing versions of NUMA-aware locks improve throughput compared with the stock version. For instance, CNA decreases the `qspinlock` overhead due to process forking and `gettimeofday()` from 48% to 32%. Meanwhile, both CST and Cohort locks improve the file system performance, as these locks scale as well as SHFLLOCKS. However, their large memory footprint starts stressing the memory allocator at higher core count, as the bottleneck completely shifts to process forking (30%). Finally, SHFLLOCKS improve performance on two fronts: they improve throughput by 1.2–1.6× while reducing the lock overhead by 35.4–95.8% at 192 threads. The significant overhead now comes from the `gettimeofday()` syscall, as perf shows almost 20% of CPU cycles.

Exim [108] is a process-intensive workload that forks a new process for every connection. Each connection then forks twice to handle messages and file system operations [77], which heavily over-subscribe the system. Exim creates about 3× copies for each message and heavily stresses the kernel in three subsystems: memory management, file systems, and network connections. On average, about 50% of the time is spent in the process forking/exiting and interrupts. Figure 5.5 (b) shows Exim throughput and memory usage with various locks. Both SHFLLOCKS and CNA improve throughput as they decrease the CPU idle time by 50% compared with CST, Cohort, and Stock while improving the useful work by ≈2%. The improvement is a result of a decrease in lock contention by 10% (relative to Stock) in the cleaning up of reverse mappings [153]. The throughput of the CST and Cohort locks decreases because these locks stress the memory allocator (see §5.1), as the benchmark generates about 8M files in 20 secs. In summary, SHFLLOCKS improve the throughput

by 1.5× compared with hierarchical locks as well as decrease the memory footprint by 40.8–92.9% among all existing versions.

Metis is an in-memory map-reduce framework, representing a page-fault-intensive workload that stresses a single lock in the kernel: the reader side of the `mmap_sem`. Figure 5.5 (c) shows that both Cohort and `CST` locks outperform all the centralized counter-based locks because of localizing the contention within a socket but at the cost of ≈80× extra memory. However, our readers-writer blocking lock is still faster than Stock because the stock version also encodes the sleeping waiters in its count indicator, as it has almost 3.4× higher atomic instructions compared with SHFLLOCK when measured with perf [154]. This workload also shows the efficiency of our under-subscribed scenario. Compared to `rwsem`, that has 33% more idle time due to its naive parking strategy, SHFLLOCK's readers do not park themselves. This results in less idle time (1.2%) and higher throughput (2.4×) than the original `rwsem`.

Summary. Figure 5.5 shows the impact of scheduling interaction, the overhead of memory allocation with respect to locks in both under- and over-subscribed cases, with varying contention levels. Our holistic design of SHFLLOCKS accommodates NUMA-awareness at high core count and shows that memory overhead (whether dynamic or static) heavily influences the scalability of applications. Compared to all locks, SHFLLOCKS reduce the memory footprint overhead up to 98.8% and 35.4% when compared with the hierarchical locks, CNA[2] [3] and Stock, respectively. The reduction stems from blocking locks, as SHFLLOCKS are 12/20 bytes in comparison to 40 bytes for CNA and Stock and ≈1.5KB for hierarchical locks.

[2] CNA only modifies spinlock, but does not affect the size of blocking locks.

[3] CNA only modifies spinlock, which does not affect the size of the blocking locks (same as the Stock version).

5.4.3 Performance Breakdown

We now do an in-depth analysis of SHFLLOCKS using a hash-table benchmark in the kernel [148]. A global lock guards the hash table. For SHFLLOCK^{NB} and SHFLLOCK^{B}, we use 1% writes, and for SHFLLOCK^{RW} (readers-writer blocking lock), we generate 1% and 50% writes on the hash table. Figure 5.6 shows the results as well as the factor analysis of SHFLLOCKS.

Non-blocking SHFLLOCK^{NB}. Figure 5.6 shows (a) throughput and (b) fairness. We calculate the fairness factor described by Dice *et al*. [136], in which we sort the number of operations performed by each thread, and divide the sum of the second half of threads' operations (sorted in increasing order) by the total number of operations. Thus, the resulting fairness factor is a number between 0.5 and 1, with a strictly fair lock yielding a factor of 0.5 and an unfair lock yielding a factor close to 1. We observe that both CNA and SHFLLOCK are the best performing, while the performance of Cohort locks is affected because of bloating of the critical section in the case of one socket. Although NUMA-aware locks impact the fairness of locks, they still maintain long-term fairness, as the fairness factor is close to 0.5.

Figure 5.6 (e) shows the improvement at 192 threads due to the various optimizations in SHFLLOCKS. Here, `Base` represents no shuffling, which behaves as the NUMA-oblivious spinlock. `+Shuffler` represents a version of SHFLLOCKS where only the very first waiter shuffles, but doesn't pass the role to other threads. This version improves the throughput by 16% over `Base`. `+Shufflers` represents the algorithm we describe in Listing 5.1, in which a shuffler passes the role to any waiter in the local socket. This approach results in almost a 10% improvement over `+Shuffler`. Finally, the `+qlast` optimization avoids the unnecessary pointer chasing done by the shuffler to determine where to insert a rellocated qnode by saving the last qnode of the threads with the same socket ID. This optimization improves throughput by 30% .

Blocking SHFLLOCK^{B}. Figure 5.6 shows (c) throughput, and (d) the fairness factor

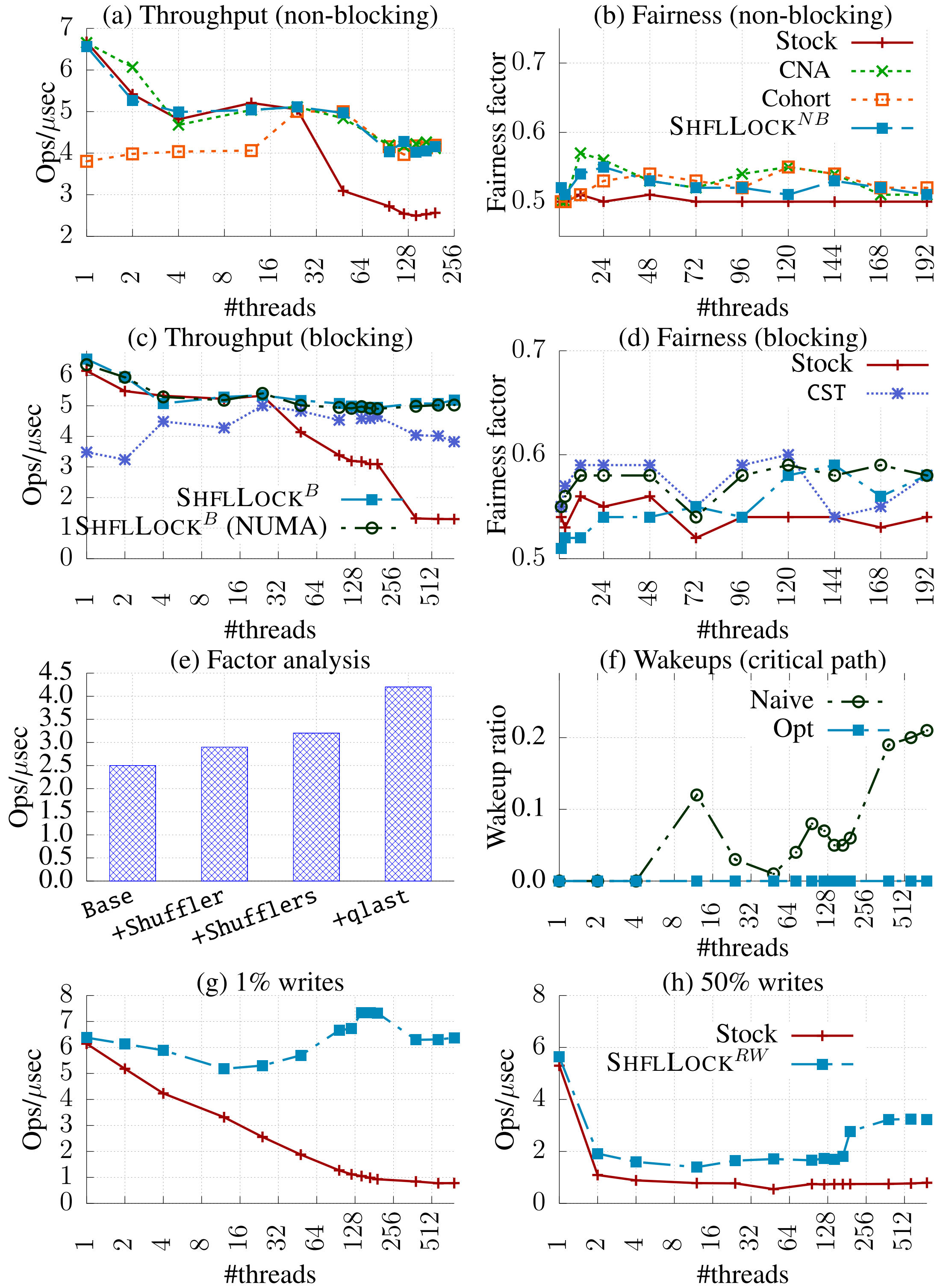

Figure 5.6: Impact on throughput and long-term fairness of non-blocking and blocking locks on the hash-table benchmark. For blocking locks, we over-subscribe the system by $4\times$. We also include the factor analysis of several phases introduced by SHFLLOCK^{NB}, and the number of wakeups in the critical path for SHFLLOCK^{B}. Later, we show the impact on throughput with centralized readers-writer locks: Stock and SHFLLOCK^{RW} for 1% and 50% writes up to $4\times$ over-subscription.

for ShflLock^B. We see that ShflLock maintains the best throughput even up to 4× over-subscription because it aggressively steals the lock in the over-subscribed case. Our lock stealing is inherently NUMA-aware because most of the remote waiters join the queue; meanwhile, the local active waiters only steal the lock if the very-next waiter (shuffler) is busy waking up its successor. We further confirm this result by only allowing the stealing from the local NUMA-socket, which shows the same throughput, as shown by ShflLock (NUMA) in the figure. Meanwhile, even in the under-subscribed scenario, we observe that the fairness factor reaches up to 0.6 because of lock stealing but does not starve waiters (d). Besides, the shuffler proactively wakes up threads that will acquire the lock soon, which completely removes the waking-up overhead from the critical path, even in the over-subscribed scenario (refer to Figure 5.6 (f)).

Blocking ShflLock^{RW}. Figure 5.6 ((g) and (h)) show that the ShflLock^{RW} has higher throughput than the stock version by 8.1× and 3.7× for 1% and 50% writes, respectively. This happens because the stock version is very inefficient, as most of the threads are blocked, resulting in idling of the CPU (99%). Meanwhile, ShflLock^{RW} maintains consistent performance regardless of the contention on the lock, even further batching readers together at a higher count to maintain good throughput. One point to note is that in the case of over-subscription, ShflLock^{RW} aggressively batches readers and writers, which slightly improves the throughput.

5.4.4 Performance With Userspace ShflLock

We now evaluate ShflLocks on three benchmarks: LevelDB for high contention, Streamcluster for the trylock interface, and Dedup for memory allocation [155]. We integrate both ShflLock and CNA into LiTL [156] for evaluation.[4] We use a set of blocking and non-blocking locks that have the best performance for the selected workloads (refer to Table 5.2).

[4]Similar to `pthread`, we use `futex()` system call to implement ShflLock^B. The waiter spins for a constant duration and then parks itself.

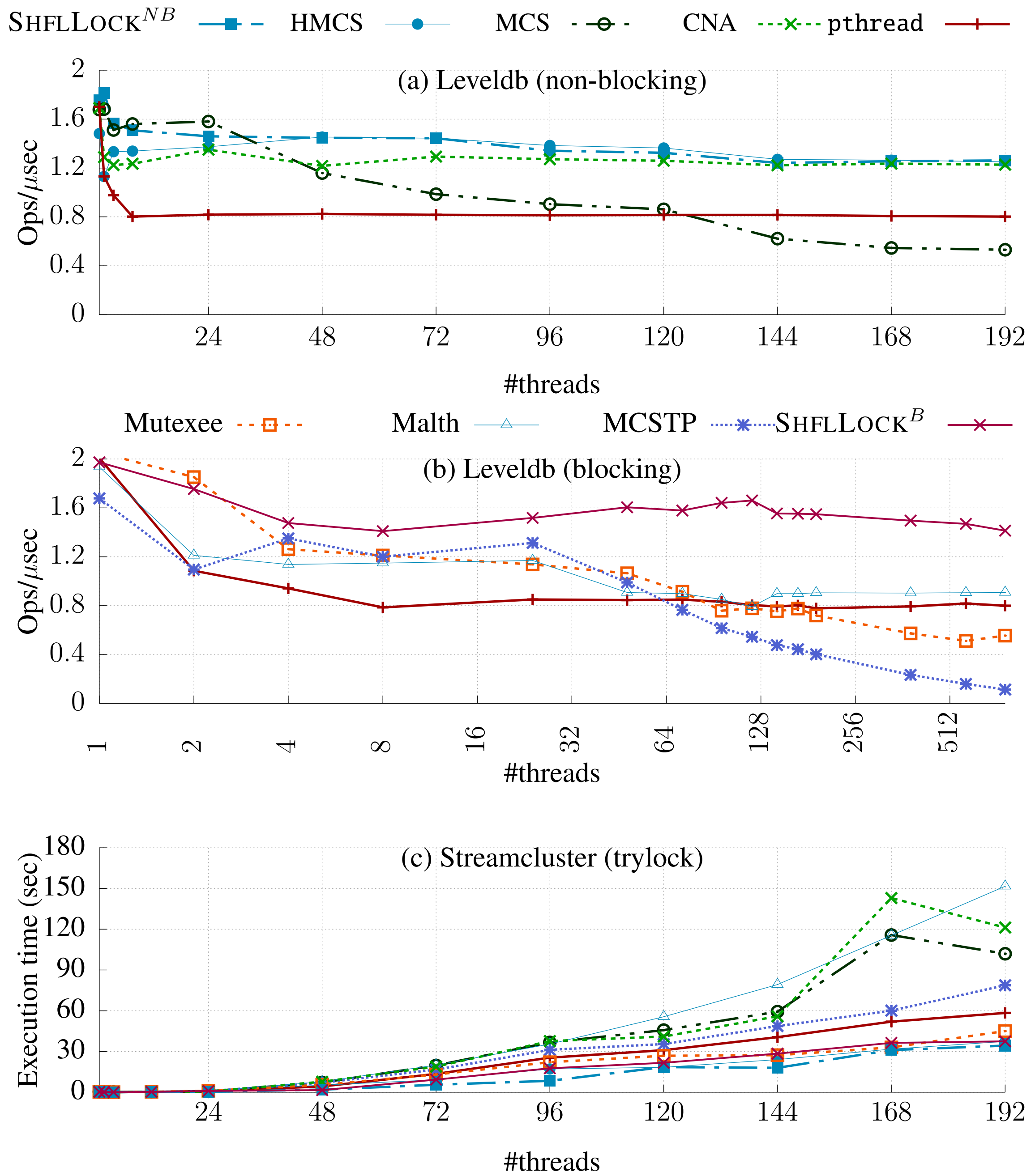

Figure 5.7: Total throughput of LevelDB benchmark and the streamcluster benchmark with various blocking and non-blocking locks. We further over-subscribe the cores for levelDB (b) to test the impact of blocking locks with 4× the number of cores.

LevelDB is an open-source key-value store [157]. We use the `readrandom` benchmark that contends on the global database lock. Figure 5.7 (a) and (b) show the throughput with non-blocking and blocking locks, respectively, with up to 4× over-subscription for the blocking

ones after running for 60 seconds. We keep `pthread` as a reference for the comparison. We find that SHFLLOCK is almost as fast as the existing NUMA-aware locks with increasing core count and is 2.4× faster than MCS locks with 192 threads. We also observe that `pthread` only scales up to eight threads because it starts parking threads. The throughput of blocking locks is better than non-blocking ones because fewer threads are contending on locks. SHFLLOCK^B outperforms others by 1.7–3.8× at 192 threads. Moreover, we see that SHFLLOCK maintains almost the same throughput even at 768 threads, and achieves 1.6–12.5× higher throughput. This happens for two reasons: efficient waking up of waiters and aggressive lock stealing, as there are still active waiters that acquire the lock.

Streamcluster is a data mining workload [137], which uses a custom barrier implementation to synchronize threads between the different phases of the application. The barrier implementation uses a mix of trylock and lock operations, as well as condition variables, which amount to almost 30% of the execution time [155]. Figure 5.7 (c) shows the execution time of streamcluster. Guerraoui *et al.* pointed out that the contention-hardened trylock interface results in better execution of this workload, which we observe for HMCS as well as for MCSTP (slightly better than MCS and CNA). However, we find that SHFLLOCKS has almost similar execution time as that of HMCS and is 1.3–4.4× faster than other locks. This happens because of our main design choice: decoupling the lock state from the waiting queue. Even though CNA is NUMA-aware, its performance is similar to MCS because the lock state and the queue tail are coupled. On further analysis, we find that queue-based locks, such as HMCS, CNA, and MCS, spend 4× extra time (failed and succeeded trylock time) and 15× excessive trylock operations than SHFLLOCK, which improves SHFLLOCK's throughout over MCS and CNA. Despite HMCS spends extra time in the trylock operation, it spends 4× less time in the lock operation than SHFLLOCK because waiters statically partition the list, which results in the most efficient NUMA-aware lock. In summary, tail and state decoupling provides a window of opportunity that allows the trylock operation to succeed.

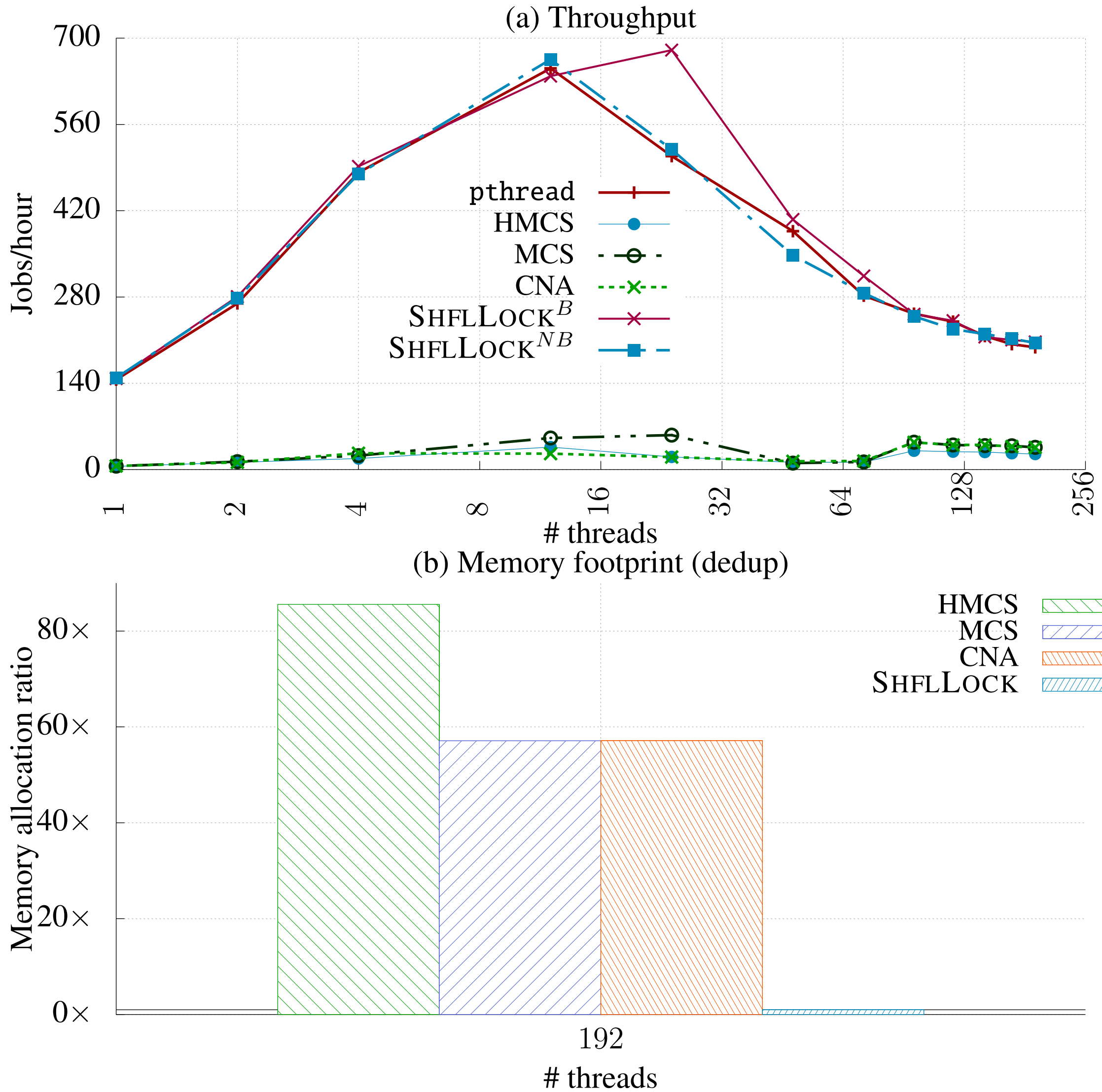

Figure 5.8: Impact of locks and their memory allocation overhead on the scalability of Dedup. We report the overall memory allocation overhead that is used during the entire run, with respect to `pthread`.

Dedup: represents an enterprise storage workload [137], which allocates up to 266K locks throughout its lifetime and heavily stresses the memory allocator as well as creates almost 3× the number of threads for several application phases. Figure 5.8 shows (a) the number of jobs per hour and (b) the ratio of the overall memory allocated during the application's lifetime with respect to `pthread`. We observe that the benchmark is not scalable after 48 cores (2 sockets) because of huge over-subscription and memory allocation. Both versions of

SHFLLOCKS have the same scalability as that of a light lock, as pthread, and the blocking version is even 5% faster at 48 cores by avoiding the lock-waiter preemption.

SHFLLOCK adds no memory overhead over pthread, but other queue-based lock add the overhead of per-thread queue nodes allocated on the heap. While these locks could theoretically allocate queue nodes on the stack, it would require application-wide changes to the Dedup code and pthread API; SHFLLOCK's queue node design is easier to deploy. In addition, the hierarchical locks also allocate per-socket structures. This leads to more than 90% of the time being spent in memory allocations. For instance, the ratio of extra memory allocation is 58–87× higher for existing queue-based locks.

5.5 Chapter Summary

Locks are still the preferred style of synchronization. However, a considerable discrepancy exists in practice and design. We classify such issues into four dominating factors that impact the performance and scalability of lock algorithms and find that none of the locks meets all the required criteria. To that end, we propose a new technique, called *shuffling*, that enables the decoupling of lock design from policy enforcement, such as NUMA-awareness or parking/wakeup strategies. Moreover, these policies are enforced entirely off the critical path by the waiters. We then propose a family of locking protocols, called SHFLLOCKS, that respects all of the factors and shows that we can indeed achieve performance without additional memory overheads.

CHAPTER 6

RELATED WORK

6.1 Ordering in Concurrency Control Algorithms

We first discuss the importance of timestamping with respect to prior research and briefly peruse invariant clocks provided by today's hardware.

Logical clocks. Logical timestamping, or a software clock, is a prime primitive for most concurrency mechanisms in the domain of concurrent programming [39, 33, 47, 50, 158, 159, 42, 110] or database transactions [5, 34, 3, 101, 100]. To achieve ordering, these applications rely on atomic instructions, such as `fetch_and_add` [47, 33, 160, 110] or `compare_and_swap` [1]. For example, software transactional memory (`STM`) [96] is an active area of research in which almost every new design [110, 160] is dependent on time-based approaches [1, 2] that use global clocks for simplifying the validation step while committing a transaction. As mentioned before, a global clock incurs cache-line contention; thus, prior studies try to mitigate this contention by applying various forms of heuristics [50, 158] or slowly increasing timers [161], or having access to a synchronized hardware clock [42]. Unfortunately, these approaches introduce false aborts (heuristics) or are limited to specific hardware (synchronized clocks). The use of `Ordo` is inspired by prior studies that either used synchronized clocks [161] or relied on specific property to achieve causal ordering on a specific architecture such as Intel X86 [42]. However, `Ordo` assumes that it has access to only invariant timestamp counters with an uncertainty window, and we expose a set of methods for the user to handle that uncertainty regardless of the architecture. Similar to `STM`, `RLU` [33] is a lightweight synchronization mechanism that is an alternative to RCU [162]. It also employs a global clock that serializes writers and readers, and uses a defer-based approach to mitigate contention. We show that even with its defer-based approach, `RLU`

suffers from the contention of the software clock, which we address with the `Ordo` primitive.

Databases rely on timestamping for concurrency control [101, 5, 3, 4, 163] to ensure a serializable execution schedule. Among concurrency control mechanisms, the optimistic approach is popular in practice [5, 101]. Yu *et al.* [100] evaluated the performance of various major concurrency control mechanisms and emphasized that timestamp-based concurrency control mechanisms such as optimistic concurrency control (`OCC`) and multi-version concurrency control mechanisms (`MVCC`) suffer from timestamp allocation with increasing core count. Thus, various `OCC`-based systems have explored alternatives. For example, `Silo` [5] adopted a coarse-grain timestamping and a lightweight `OCC` for transaction coordination within an epoch. The lightweight `OCC` is a conservative form of the original `OCC` protocol. Even though it achieves scalability, it does not scale in highly contentious workloads [164] due to its limited concurrency control. On the other hand, `MVCC` protocols still rely on logical clocks for timestamp allocation and suffer scalability collapse even in a read-only workload [100, 34].

Physical timestamping. With the pervasiveness of multicore machines, physical timestamp-based algorithms are gaining more traction. For example, `Oplog`, an update-heavy data structure library, removes contention from a global data structure by maintaining a per-core log that appends operations in a temporal order [6]. Similarly, Dodds *et al.* [43] proposed a concurrent stack that scales efficiently with the `RDTSC` counter. Likewise, quiescence mechanisms have shown that clock-based reclamation [40] eschews the overhead of the epoch-based reclamation scheme. In addition, Recbulc [165] used core-local hardware timestamps to reduce the recording overhead of synchronization events and determine global ordering among them. They used a statistical approach to determine the skew between clocks, which is around 10 cycles for their machine. Another dimension in which some of the prior works have focused is expediting record/replay [166] with the assumption of clocks are already synchronized. Unfortunately, besides Recbulc, the primary concern with these algorithms, is their assumption of access to synchronized clocks, which is not

true with any available hardware. On the contrary, all clocks have constant skew and are monotonically increasing at a constant rate, which our Ordo primitive leverages to provide a globally synchronized clock on modern commodity hardware. Thus, with the help of our API, Ordo acts as a drop-in replacement for these algorithms. The only tweak that these algorithms require is to carefully handle the case of uncertainty that is present because of these invariant clocks. The idea proposed by Recbulc is not applicable in our scenario, as it still provides a statistical guarantee, which we want to avoid for designing concurrent algorithms that want to have the property of linearizability or serializability. Moreover, our approach ensures the correct use of these clocks with the help of one-way latency.

6.2 Double Scheduling in VMs

The double scheduling phenomenon is a recurring problem in the domain of virtualization, which seriously impacts the performance of a VM. There have been comprehensive research efforts to mitigate this problem.

Synchronization primitives in VMs. Uhlig *et al.* [84] demonstrated the spinlock synchronization issue in a virtualized environment, which he addressed with synchronous hints to the hypervisor, and was later replaced by para-virtual hooks for the spinlock [83] for notifying the hypervisor to block the vCPU after it has exhausted its busy wait threshold. Meanwhile, other problems such as LWP [131], the BWW problem [120, 16], and RCU readers preemption problems were found. Gleaner [120] that addressed the BWW problem implemented a user space solution to handle tasks among a varying number of vCPUs, by manipulating tasks' processor affinity in the user space, which is difficult to maintain at runtime as it must accurately track each task launch and deletion.

From the hardware perspective, processor manufacturers added an execution control to the VMCS structure—Pause Loop Exiting (PLE) [167]– that notifies the hypervisor of the waiter via VM exit. PLE partially solves the LHP problem but can also result in false positives. Ahn et al. [168] proposed a solution on the basis of a smaller time slice to resolve

both interrupt handling and LHP-LWP problems. They proposed an LLC-based architectural solution to resolve the large overhead. This approach will result in a huge overhead for VMs with a high core count, and degradation might remain consistent.

Taebe *et al*. [86] addressed the LHP/LWP issue by exposing the time window from the hypervisor to the guest OS, which leverages this information that enables a waiter to either spin or join the waiting queue. However, their solution is not applicable to CFS [82] scheduler of Linux as it does not expose the scheduling window information. Their solution is orthogonal to our approach as we want the hypervisor to take a decision than the VM. Waiman Long [87] designed and implemented qspinlock that inherently overcomes the problem of LWP by exploiting the property of the TAS lock in the queue-based lock. It works by allowing the other waiters to steal the lock before joining the queue without disrupting the waiters' queue. However, qspinlock is still prone to LHP. Meanwhile, by annotating various locks as eCS, we confirm these problems, and further identify new sets of problems such as RP and ICP, and provide a simple solution to address the double scheduling phenomenon.

Partial handling of scheduling overhead in VMs. There have been several studies on virtualization overhead because of the software-hardware redirection [17, 16] and co-scheduling issues [116, 117, 118, 169]. For example, VMware relies on relaxed co-scheduling [116] to mitigate double scheduling problem, in which vCPUs are scheduled in batches and the stragglers are synchronized within a predefined threshold. Besides this, other works have proposed balanced vCPU scheduling [117] or even IPI based demand scheduling [118]. However, these co-scheduling approaches suffer from CPU fragmentation. On the contrary, our approach neither introduces any CPU fragmentation nor it needs to synchronize the global scheduling information for all the vCPU of a VM because each vCPU is locally penalized by the hypervisor rather than synchronizing them among other vCPUs.

Song *et al*. [115] proposed the idea of dynamically adjusting vCPUs according to available CPU resources, while allowing guest OS to schedule its tasks. They used the approach of vCPU ballooning, which avoided the problem of double scheduling and was later extended

by Cheng *et al*. [121] by designing a lightweight hotplug vCPU mechanism. Although their approach is effective in case of small VMs, it is complementary to our approach and may not scale effectively for large SMP VMs because of the overhead of migrating tasks from one vCPU to another as well as the frequent rescheduling of the targeted vCPUs. eCS, on the other hand, does not suffer from any explicit IPI and migration-specific tasks, as it only adds an overhead of a simple memory operations for a scheduling decision.

6.3 Locking Primitives

We classify prior research directions into three categories: NUMA-aware locks, runtime contention management, and timeout-based locks.

NUMA-aware locks. NUMA-aware locks address the limitation of NUMA-oblivious locks [59] by amortizing the cost of accessing the remote memory. Most of the locks are hierarchical in nature such that they maintain multiple levels of lock [63, 64, 60, 156, 61] in the form of a tree. Inspired by prior hierarchical locks [60, 61], Cohort locks [63, 64] generalized the design of any two types of locks in a hierarchical fashion for two-level NUMA machines and later extended them for the read-write locks [68]. However, neither of them addresses the memory utilization issue nor supports blocking synchronization, which leads to sub-optimal performance when multiple instances of locks are used or when the system is overloaded. Besides Cohort locks, another category of locking mechanism is based on combining [170, 171] and the remote core execution approach [172] in which a thread executes several critical sections without any synchronization. Although it outperforms Cohort locks [172], the mechanism requires application modification, which is not practical for applications with a large code base.

Contention management. The interaction between lock contention and thread scheduling determines application scalability, which is an important criterion to decide whether to spin or park a thread in an under- or over-subscribed scenario. Johnson et al. [143] addressed this problem by separating contention management and scheduling in the user space. They

use admission control to handle the number of spinning threads by running a system-wide daemon that globally measures the load on the system. Similar approaches have been used by runtimes [173] and task placement strategies inside the kernel without considering the lock subsystem [174]. Along these lines, the Malthusian lock [144], a NUMA-oblivious lock, handles thread over-subscription by randomly moving a waiter from an active list to a passive list (concurrency culling), which is inspired by Johnson et al.

Timeout-based locks. Locks with timeout capability address the problem of tolerating preemption of the threads, aborting transactions in databases or even meeting the deadline in real-time systems by abandoning their attempt to acquire the lock. Scott et al. implemented a timeout based locks [175, 142] that either modify the queue and status maintained by the lock or explicitly allocated memory for each lock acquisition. These locks are inefficient in terms of space complexity as well as do not address the cache line bouncing problem of NUMA machines. Moreover, the memory management will become a critical bottleneck for these locks with increasing core count. Cohort locks [63] also present two timeout capable locks but they implement variant of CLH lock [142], which still suffers from explicit memory management.

CHAPTER 7
REFLECTIONS

This chapter first discusses the limitations of proposed approaches. Later, we discuss some of the future works that this thesis opens up.

7.1 Limitations

We discuss the limitations of our approaches with respect to hardware timestamping, task scheduling, and dynamic lock algorithms.

7.1.1 Ordering in Concurrency with `Ordo`

The most important issue with these hardware clocks is the instruction ordering that varies with architectures. For example, in the case of `Intel` architectures, we avoided instruction reordering by relying on the `RDTSCP` instruction followed by an atomic instruction to assign the timestamp. We do a similar operation for the `ARM` architecture as well. Another important problem with the timestamp counters is that they will overflow after crossing the 64-bit boundary in the case of `Intel`, `AMD`, and `Sparc` and the 55-bit boundary for `ARM`. Even though it will take years for the counters to overflow, the algorithms should also handle this scenario like existing approaches [47].

Additionally, we assume that existing hardware clocks have constant skew and do not suffer from clock drift. However, this issue has been known to occur in various older machines [176]. On the contrary, we have not seen such an issue on any of our machines. Moreover, in a private communication, an Intel developer stated that current generations of Intel processors do not suffer from clock drift as the hardware tries to synchronize it. On the contrary, `Oplog` [6] empirically found that the clocks are synchronized. Since hardware vendors do not provide any such guarantee, our calculation is of the `ORDO_BOUNDARY` relaxes

the notion of synchronized clocks, while only assumes that they have constant skew. In addition, we believe that our algorithm (Figure 3.1) is applicable to machines with asymmetric architectures, such as AMD Bulldozer [177]. However, we could not evaluate the scalability of algorithms and the range of the `ORDO_BOUNDARY` because of the unavailability of the machine, but we believe that algorithms should still be scalable enough, as they have other bottlenecks besides timestamping.

Another important issue is further decreasing the uncertainty period. We can try to decrease this window by comparing time on the basis of thread ID, as we can expose a pairwise clock table to algorithms. However, we avoided such a method for three reasons. First, maintaining a pairwise clock table will incur memory overhead and the application has to load the whole table in the memory for reading. Second, application developers may have to resort to thread pinning so that threads do not move while comparing or doing an operation, as thread migration can break the assumptions of algorithms. Finally, the value of `ORDO_BOUNDARY` is large enough to work on multisocket machines. Moreover, `ORDO_BOUNDARY` overcomes the problem of thread migration because the cost of thread migration varies from 1–200 μs, which does not affect the correctness of existing algorithms, as the time obtained after thread migration is already greater than the `ORDO_BOUNDARY` (refer to Table 3.1). However, `Ordo` is not a panacea to solving the timestamping issue. For instance, the timestamped stack [43] is oblivious to weak forms of clock synchronization that have stutter.

7.1.2 Enlightened Critical Sections

Our e`CS` approach addresses the problem of preemptions and `BWW` in both under- and overcommitted scenarios by annotating all synchronization primitives and mechanisms in the kernel space. However, besides these primitives, kernel developers have to manually annotate a critical section if they want to avoid the preemptions while introducing their own primitives. One approach could be that the hypervisor can read the instruction pointer (IP) to figure

out an eCS, but the guest OS must provide a guest OS symbol table to resolve the IP. In addition, the current design of eCS only targets the kernel space of a guest OS, and it is still agnostic of the user space critical sections such as pthread locks. Hence, we would like to extend our approach to the user space critical sections to further avoid the preemption problem, as we believe that eCS is a natural fit for multi-level scheduling. However, we need to communicate the scheduling hint down to the lowest layer effectively, which requires designing of the eCS composability extensions.

Our annotation approach does not open any security vulnerability because our approach is based on the para-virtualized VM, and it is similar to other approaches that share the information with the hypervisor [91, 123]. By using our virtualized scheduling-aware spinning approach (eSCHDSPIN), we partially mitigate the BWW problem. However, our Hypervisor $\rightarrow$ VM methods expose scheduling information of the pCPU, but they only tell if a pCPU is overloaded or a vCPU is preempted. In addition, a VM cannot misuse this information as it will be later penalized by the hypervisor. There is also very slight possibility of priority inversion problem with our extra schedule approach. However, the window of that hypervisor-granted extra schedule is too small to incur priority inversion and performance, unlike co-scheduling approaches [117, 116] in which the scheduling window is in the order of several milliseconds.

7.1.3 Shuffling-based Lock Algorithms

Shuffling enables designing dynamic lock algorithms that can incorporate policies on the fly. This approach departs from the conventional static approach, such as hierarchical locks. However, this dynamic policy can be ineffective if shuffler is faster than waiters joining the queue. For example, in the case of NUMA-aware locks, shuffling can be ineffective, if the waiters are joining the queue after the completion of the shuffling phase. In this scenario, the SHFLLOCKS will behave as a NUMA-oblivious algorithm. We can partially mitigate this issue at the expense of regularly shuffling, i.e., a shuffler can only stop the shuffling

process on finding a node that adheres to the specified policy.

7.2 Future Work

We now outline a set of research directions that this dissertation opens up.

***Ordering*: More hardware support.** `Ordo` enables applications or systems software to correctly use invariant hardware timestamps. However, the uncertainty window can play a huge part for large multicore machines that span beyond thousands of cores. Moreover, `Ordo` only dominates after crossing the socket boundary. Hence, it is possible to further reduce the uncertainty window if processor vendors can provide some bound on the cost of cache-line access, which will allow us to use existing clock synchronization protocols with confidence, thereby decreasing the uncertainty window to overcome aborts occurring in concurrency algorithms.

***Ordering*: Algorithmic use cases.** `Ordo` is definitely beneficial to various applications, such as databases [178] and log mechanisms in file systems [38]. For instance, `Ordo` has already found its usefulness in designing a multi-version support for `RLU` [179] and its durable version [180].

***Ordering*: Rack-scale algorithms.** Another direction is designing algorithms for upcoming rack-scale architectures. In this scenario, we can devise a two-level timestamping primitive algorithms can use `Ordo` inside a machine to be multicore friendly, while using existing synchronization protocols for consistency guarantees.

***Scheduling*: Efficient n-level scheduling.** Cloud providers allow users to execute serverless functions [181, 182, 183] that may be running their own runtimes (*e.g.*, go, haskell, erlang) inside VMs. This type of execution introduces multiple layers of schedulers that are stacked on top of each other. Thus, using *e*`CS` along with scheduler activations [184] can efficiently bridge the semantic gap across schedulers.

***Locks*: Software defined synchronization.** Our shuffling mechanisms opens new opportunities to implement different policies based on the hardware behavior or the requirements of the application. For instance, until now every lock is statically compiled and is used for the entire application life cycle. This is common in the case of the OS. Thanks to the shuffling mechanism, which decouples the policy from design, we can dynamically inject various policies to change the application behavior during its life cycle. For example, we can prioritize 1) stragglers in parallel algorithms, 2) set of system calls to improve application performance, and 3) background tasks [185]. Moreover, we can modify the locking policy based on the hardware requirements, such as non-inclusive cache [135] or a multi-level NUMA-hierarchy [186].

***Locks*: Verifying and synthesizing locks.** Shuffling opens up the opportunity to verify complex lock algorithms. Because of its decoupling approach, we can leverage existing work [187] to separately verify the shuffling mechanism, which is part of the already verified MCS lock algorithm. In addition, we can synthesize or even sketch various types of locking protocols, as specified by the lock developer.

CHAPTER 8
CONCLUSION

Synchronization primitives are fundamental to the design of applications. Today's applications heavily rely on these primitives to not only improve their performance but also for correctness. Moreover, these primitives have been evolving over the past two decades to not only adopt the hardware changes for better performance but also cater to the software requirements. Unfortunately, most of the proposed primitives either fail to scale efficiently when hardware changes or require a complete redesign for software. This dissertation focuses on the scalability bottlenecks of synchronization primitives on large multicore machines. We propose several new practical synchronization primitives by utilizing either hardware or the domain knowledge of the application to improve the performance of applications.

First, we propose a new ordering primitive, called `Ordo`, that overcomes the scalability bottleneck of costly atomic instructions that do not scale on large multicore and multisocket machines. `Ordo` provides an illusion of a globally synchronized hardware clock with some uncertainty in a machine. `Ordo` leverages the hardware-provided invariant per-core clocks that increase at a constant frequency in a machine. This thesis then proposes a set of simple methods, which timestamp-based concurrency control algorithms can directly use. We show their effectiveness by applying them to five algorithms, using either physical or logical timestamp. With `Ordo`, we remove the overhead of atomic instructions and scale these algorithms across several architectures by at most an order of magnitude. `Ordo` is a fundamental ordering primitive that has a fixed cost regardless of the number of cores in a machine.

Second, this thesis evaluates the issue of various preemption problems that occurs because of synchronization primitives, as multiples of software layers are stacked together. Upon further evaluation, we find that the preemption problems are merely symptoms of a

significant problem—the double scheduling problem— in which the guest OS schedules processes on virtual CPUs, and the hypervisor schedules virtual CPUs on physical CPUs. This thesis presents a single shot solution that addresses all preemption problems arising from synchronization primitives, as well as other OS services, such as interrupt contexts processing. We use one key insight: if a certain key component of a guest OS is allowed to proceed further, the guest OS will make forward progress. We identify these critical components as synchronization mechanisms, such as spinlocks, `mutex`, `rwsem`, `RCU`, and even interrupt context. We annotate these sections with our lightweight methods and call them *enlightened critical sections* (*e*CS). Our methods expose whether a VM is executing a critical section, which the hypervisor uses to provide an extra schedule at the scheduling boundary, allowing the guest OS to progress forward.

Finally, we focus on the design of lock algorithms: the most preferred style of synchronization that almost every application uses. We study the discrepancy that exists in practice and design, which we classify into four factors that impact the performance and scalability of the lock algorithm. Moreover, none of the existing lock algorithms do meet all four factors. Hence, this thesis proposes a new technique, called shuffling, which decouples lock design from policy enforcement. Such policies are NUMA-awareness, parking/wakeup strategies, reader or writer preference in readers-writer lock design. We propose a family of locking algorithms, called SHFLLOCKS, where waiters enforce these policies, mostly off the critical path. SHFLLOCKS are the first family of lock algorithms that respect all four factors, utilize waiters without using additional memory, and achieve the best performance.

REFERENCES

[1] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proceedings of the 20th International Conference on Distributed Computing (DISC)*, Stockholm, Sweden: Springer-Verlag, Sep. 2006, pp. 194–208.

[2] T. Riegel, P. Felber, and C. Fetzer, "A Lazy Snapshot Algorithm with Eager Validation," in *Proceedings of the 20th International Conference on Distributed Computing*, ser. DISC'06, Stockholm, Sweden: Springer-Verlag, 2006, pp. 284–298.

[3] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[4] P. A. Bernstein and N. Goodman, "Multiversion Concurrency Control&Mdash;Theory and Algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, Dec. 1983.

[5] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy Transactions in Multicore In-memory Databases," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013, pp. 18–32.

[6] S. B. Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "OpLog: a library for scaling update-heavy data structures," *CSAIL Technical Report*, vol. 1, no. 1, pp. 1–12, 2013.

[7] *Xeon Processor E7-8890 v4 (60M Cache, 2.20 GHz)*, http://ark.intel.com/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz, Intel, 2016.

[8] *Data Sheet: SPARC M7-16 Server*, http://www.oracle.com/us/products/servers-storage/sparc-m7-16-ds-2687045.pdf, Oracle, 2015.

[9] Micro, *3D XPoint Technology*, 2019.

[10] Anandtech, *Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here!* 2018.

[11] ——, *The Intel Second Generation Xeon Scalable: Cascade Lake, Now with Up To 56-Cores and Optane!* 2018.

[12] P. Alcorn, *Samsung Releases New 12 Gb/s SAS, M.2, AIC And 2.5" NVMe SSDs: 1 Million IOPS, Up To 15.63 TB*, `http://www.tomsitpro.com/articles/samsung-sm953-pm1725-pm1633-pm1633a,1-2805.html`, 2013.

[13] T. P. Morgan, *Flashtec NVRAM Does 15 Million IOPS At Sub-Microsecond Latency*, `http://www.enterprisetech.com/2014/08/06/flashtec-nvram-15-million-iops-sub-microsecond-latency/`, 2014.

[14] B. Tallis, *Intel Announces SSD DC P3608 Series*, `http://www.anandtech.com/show/9646/intel-announces-ssd-dc-p3608-series`, 2015.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[16] X. Song, H. Chen, and B. Zang, "Characterizing the Performance and Scalability of Many-core Applications on Virtualized Platforms," Fudan University, Tech. Rep., 2010.

[17] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, San Jose, California, USA: ACM, 2006, pp. 2–13.

[18] Microsoft, *SQL Server 2014*, `http://www.microsoft.com/en-us/server-cloud/products/sql-server/features.aspx`, 2014.

[19] SAP, *SAP HANA 2.0 SPS 02*, `http://hana.sap.com/abouthana.html`, 2017.

[20] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, Boston, MA: USENIX Association, 2010, pp. 10–10.

[21] Intel, *Intel 64 and IA-32 Architectures Software Developer Manuals*, `https://software.intel.com/en-us/articles/intel-sdm`, 2016.

[22] Oracle, *Oracle SPARC Architecture 2011*, `http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf`, 2016.

[23] AMD, *Developer Guides, Manuals & ISA Documents*, `http://developer.amd.com/resources/developer-guides-manuals/`, 2016.

[24] ARM, *The armv8-a architecture reference manual*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html, 2016.

[25] B. C. Serebrin and R. M. Kallal, *Synchronization of processor time stamp counters to master counter*, https://patentscope.wipo.int/search/en/detail.jsf?docId=US42732522, 2009.

[26] D. Martin G., J. J. Shrall, S. Parthasarathy, and Rajesh, *Controlling time stamp counter (TSC) offsets for mulitple cores and threads*, https://patentscope.wipo.int/search/en/detail.jsf?docId=US73280125o, 2011.

[27] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[28] Intel, *Intel Xeon Processor Scalable Family Technical Overview*, https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html, 2017.

[29] ——, *An Introduction to the Intel QuickPath Interconnect*, https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009.

[30] AMD, *API NetWorks Accelerates Use of HyperTransport™ Technology With Launch of Industry's First HyperTransport Technology-to-PCI Bridge Chip*, https://web.archive.org/web/20061010070210/http://www.hypertransport.org/consortium/cons_pressrelease.cfm?RecordID=62, 2001.

[31] Intel, *The Common System Interface: Intel's Future Interconnect*, https://www.realworldtech.com/common-system-interface/5/, 2007.

[32] T. David, R. Guerraoui, and V. Trigonakis, "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013, pp. 33–48.

[33] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015, pp. 168–183.

[34] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "TicToc: Time Traveling Optimistic Concurrency Control," in *Proceedings of the 2016 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA: ACM, Jun. 2016, pp. 1629–1642.

[35] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[36] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: A Scalable Approach to Logging," in *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*, Singapore, Sep. 2010, pp. 681–692.

[37] T. Wang and R. Johnson, "Scalable Logging Through Emerging Non-volatile Memory," in *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, Hanghzou, China: VLDB Endowment, Jun. 2014, pp. 865–876.

[38] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2015, pp. 273–286.

[39] M. Arbel and A. Morrison, "Predicate RCU: An RCU for Scalable Concurrent Updates," in *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Francisco, CA, Feb. 2015, pp. 21–30.

[40] Q. Wang, T. Stamler, and G. Parmer, "Parallel Sections: Scaling System-level Datastructures," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, London, UK, Apr. 2016, 33:1–33:15.

[41] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, USA, Oct. 2012, pp. 251–264.

[42] W. Ruan, Y. Liu, and M. Spear, "Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, 40:1–40:21, Dec. 2013.

[43] M. Dodds, A. Haas, and C. M. Kirsch, "A Scalable, Correct Time-Stamped Stack," in *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India: ACM, Jan. 2015, pp. 233–246.

[44] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[45] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the Cost of Atomic Operations on Modern Architectures," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, WA, DC, USA: IEEE, Oct. 2015, pp. 445–456.

[46] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010, pp. 1–16.

[47] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-based Software Transactional Memory," in *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Salt Lake City, UT, Feb. 2008, pp. 237–246.

[48] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems," in *Proceedings of the 2012 ACM SIGMOD/PODS Conference*, Scottsdale, Arizona, USA: ACM, May 2012, pp. 61–72.

[49] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008, pp. 29–44.

[50] H. Avni and N. Shavit, "Maintaining Consistent Transactional States Without a Global Clock," in *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity*, ser. SIROCCO '08, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 131–140.

[51] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang, "BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-memory Databases," *Proc. VLDB Endow.*, vol. 9, no. 6, pp. 504–515, Jan. 2016.

[52] A. Haas, C. Kirsch, M. Lippautz, H. Payer, M. Preishuber, H. Rock, A. Sokolova, T. A. Henzinger, and A. Sezgin, *Scal: High-performance multicore-scalable data structures and benchmarks*, `http://scal.cs.uni-salzburg.at/`, 2013.

[53] B. C. Serebrin, *Fast, automatically scaled processor time stamp counter*, `https://patentscope.wipo.int/search/en/detail.jsf?docId=US42732523`, 2009.

[54] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon, "Globally Synchronized Time via Datacenter Networks," in *Proceedings of the 27th ACM SIGCOMM*, Florianopolis, Brazil: ACM, Aug. 2016, pp. 454–467.

[55] T. K. Srikanth and S. Toueg, "Optimal Clock Synchronization," *J. ACM*, vol. 34, no. 3, pp. 626–645, Jul. 1987.

[56] D. L. Mills, "A Brief History of NTP Time: Memoirs of an Internet Timekeeper," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 9–21, Apr. 2003.

[57] A. Carta, N. Locci, C. Muscas, F. Pinna, and S. Sulis, "GPS and IEEE 1588 Synchronization for the Measurement of Synchrophasors in Electric Power Systems," *Comput. Stand. Interfaces*, vol. 33, no. 2, pp. 176–181, Feb. 2011.

[58] J. Corbet, *The big kernel lock strikes again*, `https://lwn.net/Articles/281938/`, 2008.

[59] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-memory Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[60] V. Luchangco, D. Nussbaum, and N. Shavit, "A Hierarchical CLH Queue Lock," in *Proceedings of the 12th International Conference on Parallel Processing*, ser. Euro-Par'06, Dresden, Germany, 2006, pp. 801–810.

[61] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining NUMA Locks," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, San Jose, California, USA, 2011, pp. 65–74.

[62] Z. Radovic and E. Hagersten, "Hierarchical Backoff Locks for Nonuniform Communication Architectures," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03, Washington, DC, USA: IEEE Computer Society, 2003, pp. 241–, ISBN: 0-7695-1871-0.

[63] D. Dice, V. J. Marathe, and N. Shavit, "Lock Cohorting: A General Technique for Designing NUMA Locks," in *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, New Orleans, LA, Feb. 2012, pp. 247–256.

[64] M. Chabbi, M. Fagan, and J. Mellor-Crummey, "High Performance Locks for Multi-level NUMA Systems," in *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Francisco, CA, Feb. 2015.

[65] M. Chabbi and J. Mellor-Crummey, “Contention-conscious, Locality-preserving Locks,” in *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Barcelona, Spain, Mar. 2016, 22:1–22:14.

[66] S. Kashyap, C. Min, and T. Kim, “Scalable NUMA-aware Blocking Synchronization Primitives,” in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[67] J. M. Mellor-Crummey and M. L. Scott, “Scalable Reader-writer Synchronization for Shared-memory Multiprocessors,” in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’91, Williamsburg, Virginia, USA, 1991, pp. 106–113.

[68] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit, “NUMA-aware Reader-writer Locks,” in *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Shenzhen, China, Feb. 2013, pp. 157–166.

[69] R. Liu, H. Zhang, and H. Chen, “Scalable Read-mostly Synchronization Using Passive Reader-writer Locks,” in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, Jun. 2014, pp. 219–230.

[70] J. Corbet, *Big reader locks*, `https://lwn.net/Articles/378911/`, 2010.

[71] O. Nesterov, *Linux percpu-rwsem*, `http://lxr.free-electrons.com/source/include/linux/percpu-rwsem.h`, 2012.

[72] P. Zijlstra, *percpu rwsem -v2*, `https://lwn.net/Articles/648914/`, 2010.

[73] W. Long, *qspinlock: Introducing a 4-byte queue spinlock*, `https://lwn.net/Articles/582897/`, 2014.

[74] ——, *locking/mutex: Enable optimistic spinning of lock waiter*, `https://lwn.net/Articles/696952/`, 2016.

[75] Y. Liu, *aim7 performance regression by commit 5a50508 report from LKP*, `https://lkml.org/lkml/2013/1/29/84`, 2014.

[76] A. Shi, *[PATCH] rwsem: steal writing sem for better performance*, `https://lkml.org/lkml/2013/2/5/309`, 2013.

[77] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim, “Understanding Manycore Scalability of File Systems,” in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016.

[78] J. Corbet, *Introducing lockrefs*, https://lwn.net/Articles/565734/, 2013.

[79] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[80] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight Application-Level Crash Consistency on Transactional Flash Storage," in *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2015.

[81] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[82] I. Molnar, *[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]*, https://lwn.net/Articles/230501/, 2007.

[83] T. Friebel, "How to Deal with Lock-Holder Preemption," Xen Summit, Tech. Rep., 2008.

[84] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards Scalable Multiprocessor Virtual Machines," in *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, ser. VM, Berkeley, CA, USA: USENIX Association, 2004, pp. 4–4.

[85] S. Kashyap, C. Min, and T. Kim, "Scalability In The Clouds! A Myth Or Reality?" In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*, Tokyo, Japan, Jul. 2015.

[86] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont, "The Lock Holder and the Lock Waiter Pre-emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock)," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, New York, NY, USA: ACM, Apr. 2017, pp. 286–297.

[87] W. Long, *qspinlock: a 4-byte queue spinlock with PV support*, https://lkml.org/lkml/2015/4/24/631, 2015.

[88] J. Fitzhardinge, *Paravirtualized Spinlocks*, http://lwn.net/Articles/289039/, 2008.

[89] I. Molnar and D. Bueso, *Generic Mutex Subsystem*, https://www.kernel.org/doc/Documentation/locking/mutex-design.txt, 2016.

[90] I. Molnar, *Linux rwsem*, `http://www.makelinux.net/ldd3/chp-5-sect-3`, 2006.

[91] W. Long, *locking/qspinlock: Enhance pvqspinlock & introduce queued unfair lock*, `https://lwn.net/Articles/650776/`, 2015.

[92] A. Prasad, K Gopinath, and P. E. McKenney, "The RCU-Reader Preemption Problem in VMs," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA: USENIX Association, Jul. 2017, pp. 265–270.

[93] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: Bare-metal performance for i/o virtualization," in *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS XVII, London, UK: ACM, Mar. 2012, pp. 411–422.

[94] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh, "A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '15, Istanbul, Turkey: ACM, 2015, pp. 1–15.

[95] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, no. 3, pp. 146–158, 1989.

[96] N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95, New York, NY, USA: ACM, 1995, pp. 204–213.

[97] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: A Scalable Storage Manager for the Multicore Era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '09, New York, NY, USA: ACM, 2009, pp. 24–35, ISBN: 978-1-60558-422-5.

[98] H. Kimura, "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM," in *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, Melbourne, Victoria, Australia: ACM, May 2015, pp. 691–706.

[99] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, "Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks," in *Proceedings of the 1995 ACM SIGMOD/PODS Conference*, San Jose, CA: ACM, May 1995, pp. 23–34.

[100] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores," in

Proceedings of the 40th International Conference on Very Large Data Bases (VLDB), Hanghzou, China: VLDB Endowment, Nov. 2014, pp. 209–220.

[101] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance Concurrency Control Mechanisms for Main-memory Databases," in *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*, Istanbul, Turkey: VLDB Endowment, Aug. 2012, pp. 298–309.

[102] R. Zhang, Z. Budimlić, and W. N. Scherer III, "Commit Phase in Timestamp-based Stm," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08, Munich, Germany: ACM, 2008, pp. 326–335.

[103] X. Yu, *DBx1000*, `https://github.com/yxymit/DBx1000`, 2016.

[104] C. C. Minh, *TL2-X86*, `https://github.com/ccaominh/tl2-x86`, 2015.

[105] B. Lepers, V. Quéma, and A. Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," in *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2015, pp. 277–289.

[106] D. McCraken, "Object-based Reverse Mapping," *Proceedings of the Linux Symposium*, vol. 2, no. 1, pp. 1–6, 2004.

[107] *Exim Internet Mailer*, `http://www.exim.org/`, 2015.

[108] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

[109] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, Indianapolis, Indiana, USA, Jun. 2010, pp. 143–154.

[110] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira, "Type-aware Transactions for Faster Concurrent Code," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, London, UK, Apr. 2016, 31:1–31:16.

[111] Amazon, *Amazon Web Services*, `https://aws.amazon.com/`, 2017.

[112] Google, *Compute Engine*, `https://aws.amazon.com/`, 2017.

[113] VMware, *Best Practices for Oversubscription of CPU, Memory and Storage in vSphere Virtual Environments*, https://communities.vmware.com/servlet/JiveServlet/previewBody/21181-102-1-28328/vsphere-oversubscription-best-practices%5b1%5d.pdf, 2017.

[114] Q. Software, *Demystifying CPU Ready (%RDY) as a Performance Metric*, http://www.actualtechmedia.com/wp-content/uploads/2013/11/demystifying-cpu-ready.pdf, 2017.

[115] X. Song, J. Shi, H. Chen, and B. Zang, "Schedule Processes, Not VCPUs," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*, ser. APSys '13, Singapore, Singapore: ACM, 2013, 1:1–1:7.

[116] VMware, "The CPU Scheduler in VMware ESX 4.1," VMware, Tech. Rep., 2010.

[117] O. Sukwong and H. S. Kim, "Is Co-scheduling Too Expensive for SMP VMs?" In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, New York, NY, USA: ACM, Apr. 2011, pp. 257–272.

[118] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based Coordinated Scheduling for SMP VMs," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, New York, NY, USA: ACM, 2013, pp. 369–380.

[119] S. Kashyap, C. Min, and T. Kim, "Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds," in *Proceedings of the ACM SIGOPS Operating System Review*, vol. 50, Mar. 2016.

[120] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, "Gleaner: Mitigating the Blocked-waiter Wakeup Problem for Virtualized Multicore Applications," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14, Berkeley, CA, USA: USENIX Association, 2014, pp. 73–84.

[121] L. Cheng, J. Rao, and F. C. M. Lau, "vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, London, UK: ACM, Apr. 2016, 2:1–2:14.

[122] I. Molnar and D. Bueso, *Generic Mutex Subsystem*, https://www.kernel.org/doc/Documentation/locking/mutex-design.txt, 2017.

[123] G. Costa, *Steal time for KVM*, https://lwn.net/Articles/449657/, 2011.

[124] A. Kivity, *Sched: Add notifier for process migration*, https://lwn.net/Articles/356536/, 2009.

[125] V. Seeker, *Process Scheduling in Linux*, https://www.prism-services.io/pdf/linux_scheduler_notes_final.pdf, 2013.

[126] S. Kashyap, *Vbench*, https://github.com/sslab-gatech/vbench, 2015.

[127] T. A. S. Foundation, *APACHE HTTP Server Project*, https://httpd.apache.org/, 2017.

[128] Y. Mao, R. Morris, and F. M. Kaashoek, "Optimizing MapReduce for Multicore Architectures," MIT CSAIL, Tech. Rep., 2010.

[129] J. Gilchrist, *Parallel BZIP2 (PBZIP2), Data Compression Software*, http://compression.ca/pbzip2/, 2017.

[130] W. Glozer, *wrk - a HTTP benchmarking tool*, https://github.com/wg/wrk, 2017.

[131] J. Ouyang and J. R. Lange, "Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '13, Houston, Texas, USA: ACM, 2013, pp. 191–200, ISBN: 978-1-4503-1266-0.

[132] D. Matlack, *Message Passing Workloads in KVM*, http://www.linux-kvm.org/images/a/ac/02x03-Davit_Matalack-KVM_Message_passing_Performance.pdf, 2015.

[133] G. Naptov, *KVM: Add asynchronous page fault for PV guest.* https://lwn.net/Articles/359842/, 2009.

[134] D. Chinner, *Re: [regression, 3.16-rc] rwsem: optimistic spinning causing performance degradation*, https://lkml.org/lkml/2014/7/3/25, 2014.

[135] D. Mulnix, *Intel Xeon Processor Scalable Family Technical Overview*, https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, 2017.

[136] D. Dice and A. Kogan, "Compact NUMA-aware Locks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, Dresden, Germany: ACM, 2019, 12:1–12:15, ISBN: 978-1-4503-6281-8.

[137] C. Bienia, "Benchmarking Modern Multiprocessors," AAI3445564, PhD thesis, Princeton, NJ, USA, 2011, ISBN: 978-1-124-49186-8.

[138] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor," in *Proceedings of the 13th ACM*

Symposium on Operating Systems Principles (SOSP), Pacific Grove, California, USA: ACM, Oct. 1991, pp. 41–55, ISBN: 0-89791-447-3.

[139] IBM, *IBM K42 Group*, http://researcher.watson.ibm.com/researcher/view_group.php?id=2078, 2016.

[140] J. Corbet, *MCS locks and qspinlocks*, https://lwn.net/Articles/590243/, 2014.

[141] M. Chabbi, A. Amer, S. Wen, and X. Liu, "An Efficient Abortable-locking Protocol for Multi-level NUMA Systems," in *Proceedings of the 22nd ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Austin, TX, Feb. 2017.

[142] M. L. Scott, "Non-blocking Timeout in Scalable Queue-based Spin Locks," in *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, ser. PODC '02, Monterey, California, 2002, pp. 31–40, ISBN: 1-58113-485-1.

[143] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry, "Decoupling Contention Management from Scheduling," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, Mar. 2010, pp. 117–128.

[144] D. Dice, "Malthusian Locks," *CoRR*, vol. abs/1511.06035, 2015.

[145] W. Long. (2014). qrwlock: Introducing a queue read/write lock implementation, (visited on 04/15/2019).

[146] A. Blanchard, *will-it-scale*, https://github.com/antonblanchard/will-it-scale, 2013.

[147] M. Zalewski. (2017). american fuzzy lop (2.41b). http://lcamtuf.coredump.cx/afl/.

[148] J. Triplett, P. E. McKenney, and J. Walpole, "Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming," in *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, Portland, OR, Jun. 2011, pp. 11–11.

[149] B. He, W. N. Scherer, and M. L. Scott, "Preemption Adaptivity in Time-published Queue-based Spin Locks," in *Proceedings of the 12th International Conference on High Performance Computing*, ser. HiPC'05, Goa, India, 2005, pp. 7–18.

[150] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis, "Unlocking Energy," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016, pp. 393–406.

[151] D. Dice and A. Kogan, "BRAVO: Biased Locking for Reader-Writer Locks," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA: USENIX Association, Jul. 2019, pp. 315–328, ISBN: 978-1-939133-03-8.

[152] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Oct. 2017.

[153] D. McCracken, "Object-based Reverse Mapping," in *Proceedings of the Ottawa Linux Symposium*, OLS, 2004.

[154] J. Mario. (2016). C2C - False Sharing Detection in Linux Perf. `https://joemario.github.io/blog/2016/09/01/c2c-blog/`.

[155] R. Guerraoui, H. Guiroux, R. Lachaize, V. Quéma, and V. Trigonakis, "Lock—Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems," *ACM Trans. Comput. Syst.*, vol. 36, no. 1, 1:1–1:149, Mar. 2019.

[156] H. Guiroux, R. Lachaize, and V. Quéma, "Multicore Locks: The Case is Not Closed Yet," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016, pp. 649–662.

[157] S. Ghemawat and J. Dean. (2019). LevelDB, (visited on 04/20/2019).

[158] E. Atoofian and A. G. Bavarsad, "AGC: Adaptive Global Clock in Software Transactional Memory," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12, New Orleans, Louisiana: ACM, 2012, pp. 11–16.

[159] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a scalable software transactional memory," in *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'09)*, New York, NY, USA: ACM, 2009, pp. 1–10.

[160] A. Spiegelman, G. Golan-Gueta, and I. Keidar, "Transactional data structure libraries," in *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA: ACM, Jun. 2016, pp. 682–696.

[161] T. Riegel, C. Fetzer, and P. Felber, "Time-based Transactional Memory with Scalable Time Bases," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07, San Diego, California, USA: ACM, 2007, pp. 221–228.

[162] P. E. McKenney, "Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels," Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf`, PhD thesis, OGI School of Science, Engineering at Oregon Health, and Sciences University, 2004.

[163] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987, ISBN: 0-201-10715-5.

[164] K. Kim, T. Wang, R. Johnson, and I. Pandis, "ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads," in *Proceedings of the 2016 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA: ACM, Jun. 2016, pp. 1675–1687.

[165] X. Yuan, C. Wu, Z. Wang, J. Li, P.-C. Yew, J. Huang, X. Feng, Y. Lan, Y. Chen, and Y. Guan, "ReCBuLC: Reproducing Concurrency Bugs Using Local Clocks," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, Florence, Italy: IEEE Press, 2015, pp. 824–834, ISBN: 978-1-4799-1934-5.

[166] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum, "Towards Practical Default-On Multi-Core Record/Replay," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, Xi'an, China: ACM, 2017, pp. 693–708, ISBN: 978-1-4503-4465-4.

[167] M. Righini, *Enabling Intel Virtualization Technology Features and Benefits*, 2010.

[168] J. Ahn, C. H. Park, and J. Huh, "Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 394–405.

[169] 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*, ser. VEE '05, Chicago, IL, USA: ACM, 2005, pp. 13–23.

[170] P. Fatourou and N. D. Kallimanis, "Revisiting the Combining Synchronization Technique," in *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, New Orleans, LA, Feb. 2012, pp. 257–266.

[171] Y. Oyama, K. Taura, and A. Yonezawa, "Executing parallel programs with synchronization bottlenecks efficiently," in *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, 1999, pp. 182–204.

[172] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Fast and Portable Locking for Multicore Architectures," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, 13:1–13:62, Jan. 2016.

[173] G. Chadha, S. Mahlke, and S. Narayanasamy, "When Less is More (LIMO):Controlled Parallelism For improved Efficiency," in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '12, Tampere, Finland, 2012.

[174] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors," *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012.

[175] M. L. Scott and W. N. Scherer, "Scalable Queue-based Spin Locks with Timeout," in *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Snowbird, Utah, Jun. 2001, pp. 44–52.

[176] T. Gleixner, *RE: [PATCH] x86: Export tsc related information in sysfs*, `https://lwn.net/Articles/388286/`, 2010.

[177] D. Kantner, *AMD's Bulldozer Microarchitecture*, `https://www.realworldtech.com/bulldozer/9/`, 2011.

[178] N. Narula, C. Cutler, E. Kohler, and R. Morris, "Phase Reconciliation for Contended In-memory Transactions," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014, pp. 511–524.

[179] J. Kim, A. Mathew, S. Kashyap, M. K. Ramanathan, and C. Min, "MV-RLU: Scaling Read-Log-Update with Multi-Versioning," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[180] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable Transactional Memory Can Scale with Timestone," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, Apr. 2020.

[181] Amazon, *AWS Lambda*, `https://aws.amazon.com/lambda/`, 2017.

[182] Microsoft, *Azure Functions*, https://azure.microsoft.com/en-us/services/functions/, 2018.

[183] Google, *Serverless computing*, https://cloud.google.com/serverless, 2018.

[184] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, Oct. 1991.

[185] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O Path: A Holistic Approach for Application Performance," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, ser. FAST'17, Santa clara, CA, USA: USENIX Association, 2017, pp. 345–358.

[186] A. Patrizio, *HPE refreshes its Superdome servers with SGI technology*, https://www.networkworld.com/article/3236789/hpe-refreshes-its-superdome-servers-with-sgi-technology.html, 2017.

[187] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao, "Armada: Low-Effort Verification of High-Performance Concurrent Programs," in *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Online conference, Jun. 2020.