Fair and Secure Synchronization for Non-Cooperative Concurrent Systems

By

Yuvraj Patel

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: August 12, 2021

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Mikko H. Lipasti, Professor, ECE
Michael M. Swift, Professor, Computer Sciences

To
My parents – the living form of Gods
My sisters – Heny and Biji
My wife – Ankita
My children – Zeus and Yelena
My favorite animals – Mr. T and Kinky Tail (*the Mapogo lion coalition*)
My favorite gods – Lord Shiva and Lord Hanuman

A Life's Work
Titled by Andrea Arpaci-Dusseau and Remzi-Arpaci-Dusseau
Words by Yuvraj Patel (*written in Summer 2002*)

Life-previously,
Was like the dark,
Feeling all alone in,
The whole Universe.

Life-now,
Is full of beautiful Stars and Galaxies,
Glooming and Lightning,
Every moment of my life.

Life-in future,
Would see My Universe,
Full of joy and happiness,
Claiming the eternity of my soul.

Acknowledgments

First and foremost, I would like to extend my deepest gratitude to my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. This Ph.D. would not have been possible without their guidance and support. Andrea and Remzi made this journey of mine enjoyable and full of insights. Whenever I met them during the regular meetings, I would come out of the meetings as a better researcher. Their feedback has always been apt depending on the project status.

Even though they work together, both of them have the uniqueness that they bring to the table. Andrea always takes the work one level above and beyond what I have been thinking. I always tell people to talk to Andrea whenever they believe that they have done the maximum that one can ever do in their project. She will listen to you carefully and then slowly start building a better idea than what you have presented. Remzi always emphasizes on experimentation and how we should conduct experiments. I will never forget his emphasis on measure then build strategy. He is a quick-thinker and always comes up with fantastic ideas during the meeting itself.

Both Andrea and Remzi are very caring and always available. Well, that is the advantage of having two advisors (reliability - I must say.). Apart from talking about technical stuff, they always inquire about my family too. Even though their expertise is in storage systems, they al-

lowed me to work on concurrency and had trust and faith in me. This trust and faith helped me give enough confidence to sail through rough weather. They gave me a chance to teach Introduction to Operating Systems course during the summer and provided me with all the help they can. I am grateful to have chosen UW Madison and Andrea and Remzi as my advisors. Thank you, Andrea and Remzi, for being wonderful advisors.

I want to express my sincere appreciation and thankfulness to my committee members – Michael Swift and Mikko Lipasti. Both of them asked insightful questions and provided invaluable feedback for my research and dissertation. Mike has been an excellent teacher, and he has been on the papers that I have included in this dissertation. I enjoyed working with him. He spent time browsing Linux kernel code during the meetings and was always available even during the weekends for a quick chat. He helped me while I was teaching during the summer too. I hope to continue collaborating with him in the future as well. I learned a lot by attending Mikko's Advanced Architecture classes.

I was fortunate to work with a few wonderful colleagues at Hewlett Packard Labs during my internship. I would like to thank my mentors – Yupu Zhang and Daniel Gmach. I would also like to thank Kimberly Keeton, Haris Volos, and my manager Daniel Feldman for their support. I had fun working with you all on the memory manager during my internship. In particular, Yupu being an alumnus from our group, helped me a lot. I would also like to thank David Dice from Oracle Labs for his continued support and feedback. His valuable feedback has helped make our papers better. Unfortunately, he could not be part of the committee due to his other commitments. I look forward to collaborating with Dave in the near future.

I am blessed to have been part of a wonderful ADSL group at UW Madison – Thanumalayan Pillai, Lanyue Lu, Ramnatthan Alagappan,

Samer Al-Kiswany, Zev Weiss, Aishwarya Ganeshan, Tyler Harter, Suli Yang, Jun He, Sudarsun Kannan, Jing Liu, Kan Wu, Anthony Rebello, Youmin Chen, Leo Prasath Arulraj, Dennis Zhou, Eunji Lee, Chenhao Ye, Guanzhou Hu, Vinay Banakar, Akshat Sinha, Abigail Matthews, and Ruijian Huang. I would like to thank them for their feedback on my research during the group meetings and hallway discussions.

Many others have helped me either directly or indirectly during my Ph.D. I would like to appreciate the help and support received from the Computer Sciences Department and CSL staff. Angela Thorp has been extremely helpful and a go-to person to seek any information related to graduate studies. I would also like to thank the CCTAP program for providing funding for child care expenses for several years. I want to thank Cloudlab for providing an excellent environment to run experiments.

I would especially like to thank Shankar Pasupathy for being a mentor, friend, and elder brother who has constantly guided me for more than a decade. He was my mentor while I was interning at NetApp in 2007. His contributions towards my success cannot be measured in words. I would not have joined UW Madison if it was not for him. I followed his advice as he is an alumnus of the department. Thank you, Shankar, for constantly motivating me and teaching me various things in life. I will never forget your contributions, and you will have a special place in my heart forever. I would also like to thank my other colleagues at NetApp and SanDisk. I have learned a lot interacting with you all.

I would also like to extend my sincerest thanks to my professors - Shan Sundar Balasubramaniam, Sanjiv Kumar Choudhary, M.S. Radhakrishnan, Apoorva Patel, Roshan Shah, Amit Ganatra, Minesh Thaker, Tejas Patel, Divyang Pandya, Manilal Amipara, Rajiv Ranjan, Yogesh Kosta, and Ratnik Gandhi. All of you have taught me both technical and non-technical aspects that have helped me shape my life. I want to take this opportunity to express my appreciation to all my school teachers who al-

ways believed in me and constantly motivated me to aim high.

I would also like to extend my thanks to my friends in Madison – Anshul Purohit, Mohit Verma, Srinivas Tunuguntla, Amrita Roy Choudhury, Varun Chandrasekaran, Akhil Guliani, Meenakshi Syamkumar, Ram Durairajan, Akshat Sinha, Vinay Banakar, Uyeong Yang, Arjun Singhvi, Archie Abhaskumar, Vikas Goel, Rogers Jeffrey, Prabu Ravindran, and his family, Yogesh Phogat and his family. I am thankful for your kindness, friendship, and support.

I want to express my thanks to my wonderful and supportive family, who have always been there, encouraging and supporting me throughout my life. I am very grateful to have such a family – my grandmother, uncles, aunts, and cousins. They have always celebrated my success and motivated me during my failures. Even today, whenever I talk to my grandmom, she always reminds me to work hard and never relax. She continues to say these words ever since I was in high school – "Just work for a few more years, and then life will be wonderful." My uncle Sanjay and cousins Maniti, Radhika, and Jayvi have been supportive. Your presence in the family makes me relax a bit whenever I am worried about my parents' health. My late aunt Neeta and uncle Nikhil would have been happy to see me earn a Ph.D. They both have always considered me their son. It was my aunt Neeta who thought about getting me married to my wife. I want to thank you for coming up with that idea. I would also like to thank my other aunt Hema, who regularly calls my wife and inquires about my work and health. Thank you for caring and being such a constant motivation. I would also like to thank Hetal Patel for being a great friend and elder brother. You have been inspiring in many ways and will always be.

I want to express my gratitude to my father-in-law Purushottam Pendharkar for being a wonderful person. You are always ready to travel to the USA whenever we need your help. You have always trusted me and

backed my decisions. I always look up to you as a father and have great respect for you. Thank you for being part of my family.

I want to thank my two sisters – Heny and late Biji, and my brother-in-law Amit for your unconditional love, care, and unlimited support. All three of you always have supported me throughout my life and encouraged me to pursue research and chart my course. Both my sisters are a strong pillar of my life, and I have looked up to you whenever I needed help. Heny keeps a constant tab on me and always provides valuable suggestions, and forces me to think out of the box. Even though it's been almost ten years since you left Biji, I remember you every day. Immediately after my defense, everyone remembered you and joked about the iconic dance you would have performed after I defended my work. Without your persuasion, I would have never married. I want to thank my brother-in-law for being a constant source of motivation. You have known me ever since I was in school. I am in the field of Computer Science just because of you. I am 100% sure that if it were not for you, I would have been a Physicist.

I do not have enough words to thank my parents - Ajay and Anjana. I consider my parents as the living form of God and worship them every day. I know how both of you have struggled and sacrificed to make all your three kids successful. Mere words are not enough to express my love to you. You always encouraged me to take risks and embrace failures to achieve success. I can always rely upon you whenever I need help and your willingness to travel to the USA whenever I need you. You constantly inspire me to aim higher and continue to remain humble and grounded. I am incredibly indebted to my parents for their unconditional love.

I want to thank my two kids – Zeus and Yelena, for being beautiful kids. You are my two eyes through which I see the new world. Both of you would wait for me to come home so that we all can play. I am sorry for not being there to play with you many times and spend time with

you. I always look forward to the funny jokes that Zeus shares and the numerous projects that we have done together and will continue to do. It is fun to pretend-play with Yelena. You have inspired me to work on fairness aspects in this dissertation as both of you used different tricks to compete and seek attention.

Finally, one person that deserves my special heartfelt thanks is my wife – Ankita. She is a perfect partner and friend. As Ankita is not from a Computer Science background, we hardly talk about work. She has been patient enough for the past six years and has taken full responsibility to run the home smoothly. Even though you have not worked on this dissertation directly, you still are part of this dissertation in spirit. Without your help, support, trust, and love, it would have been impossible to have gotten through this six-year journey. Immediately after my defense, my mom congratulated you for supporting me and being part of my success and life for being the driving force. Thank you for being part of my life and family.

Before my Ph.D., I worked in Industry for almost nine years, designing filesystems and storage systems. I was well settled and doing great work while earning a good amount of money. By the time the majority of the students complete their Ph.D., I started my journey. Many people suggested that I rethink my decision to start Ph.D. The real reason why I pursued Ph.D. is to seek answers to a few questions that I had. I have answered a few of them, not fully answered a few of them, discarded a few of them, and added a few more new questions to my list. I have already got the prize of pursuing a Ph.D. – the pleasure of pursuing the questions. It was and will always be about the journey. Thank you, God, for everything I have.

Contents

Acknowledgments **ii**

Contents **viii**

List of Tables **xii**

List of Figures **xv**

Abstract **xxv**

1 Introduction **1**

1.1 Lock Usage 3
1.2 Scheduler-Cooperative Locks 6
1.3 Taming Adversarial Synchronization Attacks using Trāṭr . 8
1.4 Contributions 10
1.5 Overview 12

2 Background **15**

2.1 Concurrency, Synchronization & Mutual Exclusion 15
2.2 Locks 17
2.2.1 Crucial Lock Properties 18
2.2.2 Categorizing Lock Algorithms 22
2.2.3 Waiting Policy 25

2.3 Common Synchronization Primitives Implementation . . . 27
2.3.1 Pthread Spinlock . . . 27
2.3.2 Pthread Mutex . . . 29
2.3.3 Ticket Lock . . . 31
2.3.4 MCS & K42 variant . . . 32
2.3.5 Linux Kernel's Queued Spinlock . . . 37
2.3.6 Reader-Writer Lock . . . 38
2.3.7 Read-Copy Update . . . 41
2.4 Summary . . . 43

3 Lock Usage **45**
3.1 Scheduler Subversion . . . 50
3.1.1 Imbalanced Scheduler Goals . . . 50
3.1.2 Non-Preemptive Locks . . . 53
3.1.3 Causes of Scheduler Subversion . . . 54
3.2 Synchronization under Attack . . . 57
3.2.1 Synchronization and Framing Attacks . . . 59
3.2.2 Threat Model . . . 63
3.2.3 Synchronization and Framing Attacks on Linux Kernel . . . 64
3.3 Summary & Conclusion . . . 83

4 Scheduler-Cooperative Locks **85**
4.1 Lock Opportunity . . . 87
4.1.1 Inability to Control CPU Allocation . . . 87
4.1.2 Lock Opportunity . . . 90
4.2 Scheduler-Cooperative Locks . . . 91
4.2.1 Goals . . . 92
4.2.2 Design . . . 93
4.2.3 u-SCL Implementation . . . 94
4.2.4 k-SCL Implementation . . . 98

4.2.5 RW-SCL Implementation . . . 100
4.3 Evaluation . . . 102
4.3.1 Fairness and Performance . . . 103
4.3.2 Proportional Allocation . . . 107
4.3.3 Lock Overhead . . . 108
4.3.4 Lock Slice Sizes vs. Performance . . . 110
4.3.5 Real-world Workloads . . . 114
4.4 Limitations and Applicability . . . 124
4.5 Summary & Conclusion . . . 126

5 Taming Adversarial Synchronization Attacks using Trātṛ 127
5.1 Mitigating Adversarial Synchronization . . . 129
5.1.1 Existing Solutions . . . 131
5.1.2 Scheduler-Cooperative Locks . . . 132
5.1.3 Summary . . . 135
5.2 Trātṛ . . . 135
5.2.1 Goals . . . 136
5.2.2 Overview . . . 136
5.2.3 Design & Implementation . . . 138
5.3 Evaluation . . . 149
5.3.1 Overall Performance . . . 149
5.3.2 Performance of Trātṛ Components . . . 161
5.3.3 Overhead . . . 165
5.3.4 Real-World Scenarios . . . 175
5.3.5 Adding Directory Cache to Trātṛ . . . 178
5.3.6 False Positives . . . 179
5.3.7 False Negatives . . . 181
5.4 Limitations . . . 185
5.5 Summary & Conclusion . . . 186

6 Related Work 188

6.1 Lock Usage Fairness 188
6.2 Scheduler subversion 189
6.3 Adversarial Synchronization 190
6.4 Scheduler-Cooperative Locks 191
6.5 Trātṛ 193

7 Conclusions & Future Work **195**
7.1 Summary 196
7.1.1 Lock Usage 196
7.1.2 Scheduler-Cooperative Locks 198
7.1.3 Taming Adversarial Synchronization Attacks using Trātṛ 199
7.2 Lessons Learned 200
7.3 Future Work 202
7.3.1 Expand SCLs to Support Other Schedulers and Locks 203
7.3.2 SCLs in Multiple Locks Situation 203
7.3.3 Work Conserving Nature of SCLs 204
7.3.4 Scheduler-driven fairness 205
7.3.5 Analyzing Linux Kernel to find Vulnerable Data Structures 207
7.3.6 Combining SCLs and Trātṛ 208
7.3.7 Attacks on Concurrency Control Mechanisms 209
7.3.8 Opportunity-based fairness for other non-preemptive resources 210
7.4 Closing Words 211

Bibliography **212**

List of Tables

3.1 **Application & Workload details.** *The table shows the workload details for various user-space applications and the Linux kernel that we use to measure the critical section lengths.* 55

3.2 **Lock hold time (LHT) distribution.** *The table shows LHT distribution of various operations for various user-space applications and the Linux Kernel that use different data structures.* 57

3.3 **Summary of the attacks.** *Brief description of the three attacks on the Linux kernel. While the inode cache and directory cache attacks are synchronization attacks, the futex table attack is a framing attack. Note the different methods that we use to launch the attacks.* 65

4.1 **Lock Opportunity and Fairness.** *The table shows lock opportunity and the Jain fairness index for the toy example across the range of different existing locks, as well as the desired behavior of a lock.* . . 90

5.1 **Implementation summary of Trāṭṛ.** *Implementation details of the four slab-caches and three data structures that Trāṭṛ defends against the synchronization and framing attacks.* 140

5.2 **Performance of two benchmarks.** *Observed Throughput at the end of the experiment of the IC and FT benchmarks for Vanilla kernel without an attack, with an attack for Vanilla & Trāṭṛ.* 150

5.3 **Applications used for studying overhead.** *List of the applications that are part of the Phoronix test suite that we use to understand the overhead in Trāṭr.* 166
5.4 **Benchmarks used for studying overhead.** *List of the benchmarks that are part of the Phoronix test suite that we use to understand the overhead in Trāṭr.* 167
5.5 **Performance overhead study for applications.** *Comparison of performance for the various applications for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches relative to Vanilla kernel.* 169
5.6 **Performance overhead study for benchmarks.** *Comparison of performance for the various benchmarks for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches.* 170
5.7 **Applications used to study kernel threads overhead.** *List of the applications that are part of the Phoronix test suite used for measuring the impact of kernel threads.* 172
5.8 **Memory overhead study for the applications.***Comparison of the memory overhead for various applications for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches. The numbers in the bracket in the last column show the % increase in the total memory allocated to all the slab caches.* 173
5.9 **Memory overhead study for the benchmarks.***Comparison of the memory overhead for various benchmarks for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches. The numbers in the bracket in the last column shows the % increase in the total memory allocated to all the slab caches.* 174
5.10 **Real-world scenarios study with three real-world applications.** *List of the real-world applications and their workloads used for understanding how Trāṭr performs in real-world scenarios.* . . . 175

5.11 **Real-world scenario description.** *List of three real-world scenario summary and resource allocation to each container in each of the scenario.* 175

List of Figures

3.1 **Scheduler subversion with UpScaleDB.** *We use a modified benchmarking tool ups_bench available with the source code of UpScaleDB to run our experiments. Each thread executes either find or insert operations and runs for 120 seconds. All threads are pinned on four CPUs and have default thread priority. "F" denotes find threads while "I" denotes insert threads. "Hold" represents the critical section execution, i.e., when the lock is held; "Wait + Other" represents the wait-times and non-critical section execution. The value presented on the top of the dark bar is the throughput (operations/second).* 51

3.2 **Performance of Exim Mail Server under inode cache attack.** *Throughput and Average Latency timeline of Exim Mail Server when under inode cache attack. Prepare to attack means that the attacker starts to launch the attack and initiates probing to break the hash function and identify the superblock address. Upon identifying the superblock address, the attacker can target a hash bucket and launch an attack.* 68

3.3 **Internal state of inode cache when under inode cache attack.** *The graph present an overall picture of the inode cache when an attacker is launching the attack. In particular, the timeline shows the lock hold times, the cumulative wait times to acquire the inode cache lock, and the maximum number of entries of the victim and the attacker in the inode cache. The victim is running the Exim Mail Server.* 71

3.4 **Performance of UpScaleDB under futex table attack.** *Throughput and Average Latency timeline of UpScaleDB when under futex table attack. Prepare to attack means that the attacker starts to launch the attack and initiates probing to identify the hash bucket that UpSCaleDB uses. After identifying the hash bucket, the attack is launched by spawning thousands of threads and parking them in the identified hash bucket.* 74

3.5 **Internal state of futex table when under futex table attack.** *The graphs present an overall picture of the futex table when an attacker is launching the attack. In particular, the timeline shows the lock hold times, the cumulative wait times to acquire the hash bucket lock, and the maximum number of entries of the victim and the attacker in the futex table. The victim is running UpScaleDB.* 76

3.6 **Performance of Exim Mail Server under directory cache attack.** *Throughput timeline of Exim Mail Server when under directory cache attack. There is no need to prepare for the attack with directory cache attack. The attack starts immediately by creating millions of negative dentries.* 79

3.7 **Internal state of dentry cache when under dentry cache attack.** *The graphs present an overall picture of the dentry cache when an attacker is launching the attack. In particular, the timeline shows the lock hold times for the read-side critical section, the time taken to complete synchronize_rcu() call, and the maximum number of the entries of the victim and attacker in the dentry cache. The victim is running the Exim Mail Server.* 82

4.1 **Impact of Critical Section Size.** *The behavior of existing locks when the critical section sizes of two threads differ. The CFS scheduler is used, and each thread is pinned on a separate CPU. "wait" represents the time spent waiting to acquire the lock; "hold" represents the critical section execution, i.e., the time the lock is held; "other" represents the non-critical section execution.* 88

4.2 **User-space Scheduler-Cooperative Locks.** *The lock is shown as a dashed box, and each lock acquisition request is shown as a node (box). The arrow represents the pointer to the next node that forms the queue. "R" indicates running, and the lock is owned. "W" indicates waiting for the slice. In (1), the lock is initialized and free. (2) A single thread A has acquired the lock. The tail "T" pointer points to itself and the next "N" pointer points NULL. (3) Thread B arrives to acquire the lock and is queued. As B is the next-in-line to acquire the lock; it spins instead of parking itself. The tail and the next pointers point to B. (4) Thread A releases the lock, but B will wait for its turn as the lock slice has not expired. (5) Thread C also arrives to acquire the lock and is queued after B. C parks itself as it is not the next-in-line to acquire the lock. The tail now points to the C as it is the last one to request lock access. (6) Thread A again acquires the lock as it is the owner of the lock slice. (7) Thread A releases the lock. (8) A's lock slice is over, and B is now the owner of the slice. C is woken up and made to spin as it will acquire the lock next. The tail and the next pointers now point to C. (9) A again tries to acquire the lock but is penalized and therefore will wait for the penalty period to be over before it can be queued.* 95

4.3 **Reader-Writer Scheduler-Cooperative Locks.** *The dashed box represents the lock. "N" represents the value of the counter. "READSLICE" and "WRITESLICE" represents the read and write slice that the lock is currently in. In (1) The lock is initialized and free. (2) A reader thread R1 acquires the lock and continues its execution in the critical section. (3) Another reader thread, R2, also joins in. (4) Reader R1 leaves the critical section. (5) A writer thread W1 arrives and waits for the write slice to start. (6) Write slice is started, and W1 waits for reader R2 to release the lock. (7) Reader R2 releases the lock and the writer W1 acquires the lock. (8) Reader R1 now arrives again and waits for the read slice to start. (9) W1 releases the lock, and the read slice starts allowing the reader R1 to acquire the lock.* . 100

4.4 **Comparison on 2 CPUs.** *The graphs present a comparison of four locks: mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for 2 (a and b) and 16 (c and d) threads, each has the same thread priority. For 2 threads, each thread has a different critical section size (1 µs vs. 3 µs). For 16 threads, half have shorter critical section sizes (1 µs) while others have a larger critical section size (3 µs). "TG" stands for the thread group.* 104

4.5 **Comparison on 16 CPUs.** *The graphs present a comparison of four locks: mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for 2 (a and b) and 16 (c and d) threads, each has the same thread priority. For 2 threads, each thread has a different critical section size (1 µs vs. 3 µs). For 16 threads, half have shorter critical section sizes (1 µs) while others have a larger critical section size (3 µs). "TG" stands for the thread group.* 106

4.6 **Changing Thread Proportionality.** *Comparison of the four locks: mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for four threads running on two CPUs having different thread priorities (shown as ratios along the bottom) and different critical section sizes. The number on the top of each bar shows the lock usage fairness.* 107

4.7 **Lock Overhead Study.** *The figure presents two lock overhead studies. On the left(a), the number of threads and CPU cores are increased, from 2 to 32, to study scaling properties of u-SCL and related locks. We pin each thread on a separate CPU. On the right(b), the number of CPUs is fixed at two, but the number of threads is increased from 2 to 32.* 109

4.8 **Impact of lock slice size on performance.** *The top figure (a) shows the throughput across two dimensions: critical section and the slice size. The bottom figure (b) shows the wait-time distribution when the lock slice varies, and the critical section size is 10 µs.* . . . 111

4.9 **Interactivity vs. Batching.** *The figure shows the comparison of the wait-time to acquire the lock for mutex, spinlock, ticket lock and u-SCL.* 113

4.10 **Mutex and u-SCL performance with UpScaleDB.** *The same workload is used as Section 3.1.1. The same CFS scheduler is used for the experiments. "F" denotes find threads while "I" denotes insert threads. The expected maximum lock hold time is shown using the dashed line. "Hold" represents the critical section execution, i.e., the time until the lock is held; "Wait + Other" represents the wait-times and non-critical section execution. The number on top of the dark bar represents the throughput (operations/second). The left figure (a) shows the same graph as shown in Section 3.1.1. The right figure (b) shows the performance of u-SCL.* 115

4.11 **Comparison of RW-SCL and KyotoCabinet** *The dark bar shows the lock hold time for each individual thread and the light bar shows the lock opportunity not being unused. The values on top of the bar shows the aggregated throughput (operations/sec) for the writer and reader threads.* 117
4.12 **Performance of RW-SCL with reader and writer scaling.** *For reader scaling, only one writer is used while for writer scaling, only one reader is used. The number of readers and writers vary for reader scaling and writer scaling experiments. The dark bar shows the lock hold time while the light bar shows the unused lock opportunity. The values on top of the bar shows the throughput of the writers and readers.* 119
4.13 **Rename Latency.** *The graph shows the latency CDFs for SCL and the mutex lock under the rename operation. The dark lines show the distributions for the long rename operation (the bully), whereas lighter lines represent the short rename operation costs (the victim). Dashed lines show standard mutex performance, whereas solid lines show k-SCL performance.* 120
4.14 **Rename lock performance comparison.** *The figure presents a comparison of two locks – mutex and k-SCL for 2 threads on two CPUs; each has the same thread priority, and one thread is performing rename operations on a directory that is empty while another thread is performing rename on a directory having a million empty files.* 122
4.15 **k-SCL inactive thread detection.** *Timeline showing inactive threads detection and how the latency of the bully program varies depending on the number of active threads.* 123

5.1 **IC benchmark performance comparison for spinlock and k-SCL in Linux kernel.** *Timeline of the throughput showing the impact due to the attack for the Vanilla kernel having spinlock and kernel having k-SCL under the inode cache attack.* 134

5.2 **High-level design of Trātṛ.** *Design showing the four mechanisms of Trātṛ. The Tracking and Prevention mechanisms are part of slab-cache management. Layer 1 Detection measures the synchronization stalls to indirectly measure long critical sections. On finding longer stalls, Trātṛ triggers layer two checks if a user has a majority of object allocations. On finding one, Trātṛ identifies that user as an attacker and initiates prevention and recovery mechanisms. The prevention mechanism prevents the attacker from allocating more entries. Depending on the type of data structure, an appropriate recovery is initiated. The upper box shows the common code, while the lower box shows data structure-specific code.* 138

5.3 **Flowchart of the kernel thread associated with a data structure.** *The kernel thread that is associated with a data structure performs the detection and recovery mechanism. As part of the detection mechanism, the thread probes the synchronization primitives. If a synchronization stall is more than the threshold, appropriate action is initiated. Upon detecting an attack, the thread executes the TCA check to identify the attacker and initiate prevention window. Lastly, the thread initiates the recovery of the data structure.* 141

5.4 **Probing window behavior under normal conditions.** *Under normal conditions without an attack, during the probing window, after probing a lock once, the kernel threads sleep for 5 to 20 milliseconds. Note that the size of the probing window is randomly chosen.* . 143

5.5 **Probing window behavior when under attack.** *When an attack is ongoing, during the probing window, upon identifying that one synchronization stall is more than the threshold limit, Trātṛ dynamically increases the probing window size and aggressively probes the synchronization primitive to detect an attack early.* 144

5.6 **IC benchmark performance without attack, with attack and with Trātṛ.** (*a*) *Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trātṛ kernel. With Trātṛ, the attacker is not able to launch an attack.* (*b*) *Timeline of the average latency observed every second while creating the files.* 152

5.7 **Internal state of inode cache when under inode cache attack.** *The graphs present an overall picture of the inode cache when an attacker is launching the attack. In particular, the timeline shows the lock hold times of the attacker, the cumulative wait times to acquire the inode cache lock, and the maximum number of entries of the attacker for the Vanilla kernel and Trātṛ. The victim is running the IC benchmark.* 154

5.8 **FT benchmark performance without attack, with attack and with Trātṛ.** (*a*) *Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trātṛ kernel.* (*b*) *Timeline of the average latency for another experiment of the time to release the lock.* 157

5.9 **Internal state of futex table when under futex table attack.** *The graphs present an overall picture of the futex table when an attacker is launching the attack. In particular, the timeline shows the lock hold times of the victim, the cumulative wait times to acquire the hash bucket lock, and the maximum number of entries of the attacker. The victim is running FT benchmark.* 159

5.10 **Performance of Trāṭr components.** *Throughput timeline for the futex and inode cache attacks explaining the importance of detection, prevention and recovery. We also show the timeline for Trāṭr-TDP. Trāṭr-TDP denotes the kernel version that has the tracking, detection and prevention mechanisms enabled.* (*a*) *For Trāṭr-TDP, as there is no recovery mechanism enabled, the FT benchmark observes a significant drop in the performance.* (*b*) *However, for IC benchmark, the prevention mechanism prevents the attacker from expanding the hash bucket leading to similar performance as Trāṭr.* 162

5.11 **Performance comparison of the various applications across different scenarios.** *Performance comparison of Vanilla and Trāṭr with and without attack when subjected to different attacks.* (*a*) *&* (*b*) *shows the performance when multiple applications run within a single machine for futex table and inode cache attack.* (*c*) *shows the ability of Trāṭr to handle simultaneous attacks.* 177

5.12 **Performance comparison of the Exim mail server when under dcache attack.** *Performance comparison of Vanilla and Trāṭr with and without attack when subjected to directory cache attack. Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trāṭr kernel.* 178

5.13 **Performance comparison of victim when a defense aware attacker launches inode cache attack.** (*a*) *Timeline of the throughput showing the impact on the throughput due to the defense-aware attacker.* (*b*) *Lock hold times comparison to highlight how the defense-aware attacker remains under the threshold limits to evade detection.* 182

5.14 **Performance comparison of victim when a defense aware attacker launches futex table attack.** *Timeline of the throughput showing the impact on the throughput due to the defense-aware attacker.* 184

Abstract

In shared environments such as operating systems, servers (databases, key-value stores), and hypervisors, multiple tenants with varied requirements compete to access the shared resources, making strong performance isolation necessary. Locks are widely used synchronization primitives that provide mutual exclusion in such environments. This dissertation focuses on the concurrency issues arising from sharing synchronization primitives amongst multiple tenants.

Shared data structures change as multiple tenants execute their workloads. These state changes can lead to a situation where the critical section sizes vary significantly. In this dissertation, we ask the following question: what happens to performance and fairness when a single lock protects variable-sized critical sections?

To answer the above question, in the first part of this dissertation, we introduce the idea of lock usage that deals with the time spent in the critical section. We then study unfair lock usage in benign and hostile settings. For benign settings, unfair lock usage can lead to a new problem called scheduler subversion where the lock usage patterns determine which thread runs instead of the CPU scheduler. In a hostile setting, unfair lock usage can lead to a new problem we call adversarial synchronization. Using Linux containers, we introduce a new class of attacks – synchronization attacks – that exploit kernel synchronization to harm

application performance. Furthermore, a subset of these attacks – framing attacks – persistently harm performance by expanding data structures even after the attacker quiesces. We demonstrate three such attacks on the Linux kernel, where an unprivileged attacker can target the inode cache, the directory cache, and the futex table.

In the second part of the dissertation, to mitigate scheduler subversion, we introduce Scheduler-Cooperative Locks (SCLs), a new family of locking primitives that control lock usage and aligns with system-wide scheduling goals; our initial work focuses on proportional share schedulers. Unlike existing locks, SCLs provide an equal or proportional time window called lock opportunity within which each thread can acquire the lock one or more times. We design and implement three different SCLs – a user-level mutex lock (u-SCL), a reader-writer lock (RW-SCL), and a simplified kernel implementation (k-SCL). Our evaluation shows that SCLs are efficient and achieve high performance with minimal overhead under extreme workloads. We port SCLs in two user-space applications (UpScaleDB and KyotoCabinet) and the Linux kernel to show SCLs can guarantee equal or proportional lock opportunity regardless of the lock usage patterns.

In the third part of the dissertation, to mitigate adversarial synchronization, we design Trātṛ, a Linux kernel extension. Trātṛ can detect and mitigate synchronization and framing attacks with low overhead, prevent attacks from worsening, and recover by repairing data structures to their pre-attack performance. Trātṛ comprises four mechanisms: tracking, detection, prevention, and recovery. Our evaluation shows that Trātṛ can detect an attack within seconds, recover instantaneously while guaranteeing similar performance to baseline, and detect simultaneous attacks.

1 Introduction

The last decade has seen a phenomenal change in the hardware arena, where computer architects focus on adding more cores to the machines instead of increasing CPU frequency [65, 137]. This change is primarily driven by the physical limitations of semiconductor-based microelectronics to reduce the size of the individual components. Therefore, software developers have to rethink their applications to leverage the multicore machines instead of relying on application performance increasing due to an increase in the CPU frequency.

While leveraging these multicore machines to build concurrent systems, developers focus on embracing concurrency [76, 146]. When multiple events occur concurrently, certain interleavings are not desired and can lead to incorrect behavior. Thus, care has to be taken to always synchronize the possible interleavings to avoid undesirable behavior.

Synchronization is an act of handling the concurrent events atomically and by ensuring the ordering of events properly. Various synchronization primitives have been designed by researchers and practitioners for decades [10, 16, 26, 28, 37, 38, 45, 48, 50, 53, 56, 57, 62, 64, 73, 74, 85, 86, 101, 102, 109, 119, 131, 133, 135, 164].

Mutual exclusion is one of the ways to implement atomicity where a critical section needs to be executed atomically to avoid undesirable behavior. A critical section is part of the program accessing shared data and hence needs to be protected by synchronization primitives. Locks are one

of the most common options to enforce mutual exclusion. However, it is not always easy to design concurrent systems, and many concurrent systems suffer from bottlenecks [14, 18, 31, 49, 50, 57, 69, 82, 96, 99, 106, 107, 124, 148, 162].

Designers of locking algorithms keep several properties in mind – safety, liveness, fairness, and performance while designing a locking algorithm [138]. Guaranteeing fairness to all the participating entities is one of the key properties. Lock fairness is a well-studied problem, and many solutions are available that provide fairness properties [45, 53, 73, 90, 102, 109, 139, 147, 150]. These solutions have been designed keeping in mind that the locks will be used in a cooperative environment where the locks are used within the same program. Any side effect of a possible unfair behavior will be observed within the program itself.

Shared infrastructure is becoming common in data centers and cloud computing environments where multiple tenants run on the same physical hardware. In these environments, tenants having varied requirements can compete to access the resources and thus place a heavy burden on system software to isolate mutually distrusting tenants. Multiple solutions such as Eyeq, IceFS, cgroups, and Pisces are available to provide strong performance isolation so that the behavior of one tenant cannot harm the performance of other tenants [47, 72, 81, 100, 112, 143, 158].

This dissertation focuses on the locks within the shared environments and the concurrency problems that arise due to the sharing. One of the implicit assumptions while offering lock fairness is that the critical sections are similar in size. However, this assumption may be broken in the shared environments where one tenant may entirely have different requirements than the other. When data structures grow or shrink due to varying workloads, the critical section will likely change depending on the state of the data structure.

In this dissertation we ask the question – what is the impact on perfor-

mance and fairness properties when a single lock protects variable-sized critical sections? In particular, we question if the critical section can increase in size naturally in due course or during scenarios where a malicious actor can artificially increase the size. We answer the question by introducing a new lock property – lock usage, that is lacking in the previous locking algorithms [123]. Lock usage is the amount of time spent in the critical section while holding the lock.

This dissertation has three parts to it. In the first part, we introduce lock usage and understand the impact of unfair lock usage in benign and hostile settings. In doing so, we present two new problems – scheduler subversion [123] and adversarial synchronization [121] – that can lead to poor performance and denial-of-service. Scheduler subversion is an imbalance in the CPU allocation when the lock usage pattern dictates the CPU allocation instead of the CPU scheduler. In adversarial synchronization, a malicious actor attacks the synchronization primitives to harm application performance.

In the second part, we address the scheduler subversion problem by designing Scheduler-Cooperative locks, a new family of locking algorithms that can guarantee lock usage fairness [123]. Finally, in the third part, we address the problem of adversarial synchronization by designing Trāṭṛ[1], a new Linux extension that can quickly detect and mitigate attacks on the synchronization primitives [121].

1.1 Lock Usage

Locks are an integral component of any concurrent system. Locks are used to ensure mutual exclusion, and hence they should ensure specific properties such as safety, liveness, and fairness while having low overhead to enable frequent access.

[1]Trāṭṛ in Sanskrit means a guardian or a protector. It is pronounced as Traa + tru

Amongst all the properties, fairness offers the strongest guarantees as it ensures a static bound before a thread can acquire a lock and make forward progress. Generally, the static bound is defined by the order in which the threads acquire the locks. By doing so, no thread can access the lock twice while some other threads are kept waiting. We call this type of fairness *lock acquisition fairness*.

In the first part of the dissertation, we show that a critical lock property called lock usage is missing. We define lock usage as the amount of time spent in the critical section while holding the lock. For example, if two threads are trying to access a lock that protects a linked list, a thread that traverses the linked list is going to spend more time in the critical section compared to another thread that always inserts an entry at the head of the linked list.

When one or more threads continue to spend more time in the critical section than other threads, an imbalance created in the lock usage can lead to two problems – performance and security. A data structure continuously undergoes state changes depending on the workload that will eventually grow or shrink a data structure. Therefore, as the data structure state changes, there will be a variation in the critical section sizes that the thread executes.

The problem of lock usage becomes crucial in shared environments such as a server (databases, key-value stores, etc.), operating system, or hypervisor, where multiple tenants are executing a variety of applications and workloads. In such environments, multiple tenants are accessing the same shared infrastructure that is protected by various synchronization primitives such as locks, reader-writer locks, and RCU.

A new problem related to performance occurs due to lock usage imbalance, when a thread that spends more time in the critical section is allocated an equivalent CPU time compared to another thread that spends less time in the critical section. Although the CPU scheduling goals may

have been to allocate each thread an equal share of CPU, both the threads may not receive equal share leading to an imbalance in the CPU scheduling goals. We call this new problem scheduler subversion and is found in a benign setting where tenants compete against each other to access the shared locks but are not malicious.

Scheduler subversion is an imbalance caused in the CPU allocation, and it arises when instead of the CPU scheduler determining the proportion of the CPU each thread obtains, the lock usage pattern dictates the share. Using a real-world application – UpScaleDB, we illustrate the problem and then explain what conditions can lead to scheduler subversion problem. We observe that lock contention happens when threads spend more time in the critical section that can significantly vary in size.

Another problem related to security occurs in a hostile setting where tenants are competing to access the shared locks and may be malicious. In such a setting, a malicious actor can artificially grow the data structure to create performance interference via adversarial synchronization. Once the data structure grows, the critical section size will be longer; if the malicious actor holds the lock longer, the victims will have to wait longer to acquire the lock causing artificial lock contention, leading to poor performance. We call this a synchronization attack.

If victims happen to access the expanded data structure, they will be forced to spend more time in the critical section. Other victims will have to now wait longer to acquire the lock too. We call this a framing attack where the malicious actor forces the victim to spend more time in the critical section and make other victims wait to acquire the lock leading to poor performance.

We show three different attacks on the Linux kernel where an unprivileged malicious actor can target three different vulnerable kernel data structures leading to poor performance and denial-of-service using containers. Even though the goal of the malicious actor is to target the syn-

chronization primitives, the method applied to launch an attack is different. It signifies that there is more than one way to launch synchronization and framing attacks.

Even though containers provide significant isolation through a variety of techniques, we show that synchronization and framing attacks can bypass those isolation techniques making the victims suffer economically while observing poor performance and denial-of-service.

Therefore, it becomes crucial to consider lock usage as an essential property to support in shared environments where multiple tenants can execute various applications and workloads. Furthermore, to guarantee strong performance isolation in such environments, it is necessary to look at lock usage.

1.2 Scheduler-Cooperative Locks

In the second part of this dissertation, we continue looking into the problem of scheduler subversion. Scheduler subversion arises when the threads spend most of their time in the critical section, and the critical section sizes vary significantly. Once these two conditions hold, there is an imbalance in the lock usage leading to scheduler subversion. In this part, we focus on addressing the problem of scheduler subversion by building locks that align with the scheduling goals.

As discussed earlier, in existing applications, the critical section size varies when the data structure grows. The critical section size is not a static property and continues to change dynamically depending on the data structure state. Therefore, to remedy the problem of scheduler subversion, we define the concept of lock usage fairness.

Lock usage fairness ensures that each competing thread receives an equal or proportional opportunity to use the lock. To quantify the lock usage fairness, we introduce a new metric – lock opportunity. Lock op-

portunity is defined as the amount of time a thread spends holding a lock or acquiring the lock because the lock is available. As each thread is guaranteed a dedicated window of opportunity, no thread can dominate the lock usage leading to an imbalance in the scheduling goals.

Using a simple example, we show how existing locks – a simple spinlock, a pthread mutex, and a ticket lock – cannot control the CPU allocation. In all these cases, we observe that these locks have a very low lock usage fairness score. Even though a ticket lock ensures lock acquisition fairness, it still has a low lock usage fairness index. Therefore, a need arises to design a new locking primitive where lock usage and not just the lock acquisition determines the lock ownership.

Building upon the idea of lock usage fairness, we design Scheduler-Cooperative Locks (SCLs) – a new family of locks that align with the CPU scheduling goals and ensures lock usage fairness. The key components of an SCL lock are lock usage accounting, penalizing the threads depending on the lock usage, and dedicated lock opportunity using lock slice. SCLs provide a dedicated window of opportunity to all the threads. Thereby, SCLs are non-work-conserving relative to state-of-the-art locks.

Lock usage accounting helps with the lock usage tracking of all the threads participating in the lock acquisition process. The accounting information can then help identify the dominant threads, and they can be penalized appropriately for using their full quota of lock usage. By penalizing the dominant threads, SCLs present an opportunity to other threads to acquire the lock. Lastly, to avoid excessive locking overheads, SCLs use lock slices to allow a window of time where a single thread is allowed to acquire or release as often as it would like. Threads alternate between owning the lock slices, and thus one can view that lock slices virtualizes the critical section to make the threads believe that it has the lock ownership to itself.

Using these three components, we implement three different types of

SCLs – a user-space SCL (u-SCL), a reader-writer SCL (RW-SCL), and a simple kernel implementation (k-SCL). We use existing lock implementations to implement these three SCLs and extend them to support lock usage fairness. It shows that it is easy to extend the existing locking algorithms to support lock usage fairness. Furthermore, the implementations use existing and new optimization techniques to improve efficiency and lower the overhead.

We experimentally show that SCLs can achieve the desired behavior of allocating CPU resources proportionally in a variety of synthetic lock usage scenarios. We also show that SCLs have low overhead and can scale well. We also demonstrate the effectiveness of SCLs by porting SCLs in two user-space applications (UpScaleDB and KyotoCabinet) and the Linux kernel. In all three cases, regardless of the lock usage patterns, SCLs ensure that each thread receives proportional lock allocations that match those of the CPU scheduler.

1.3 Taming Adversarial Synchronization Attacks using Trātṛ

In the third part of this dissertation, we shift the focus from the benign settings in a shared environment to hostile settings. We now look into how we can address the security aspects of lock usage fairness when a malicious actor can deliberately grow the data structures to launch synchronization and framing attacks.

Using the three different synchronization and framing attacks on three different data structures in the Linux kernel, we illustrate there is no single way to launch the synchronization and framing attacks. Instead, an attacker has multiple options to launch these attacks and can launch these attacks simultaneously. Even worse, with framing attacks, the attacker does not have to participate once the data structure has grown signifi-

cantly. Thus, a malicious actor can inflict severe performance and economic impact on the victims using these attacks.

Attacks on synchronization primitives can be addressed by interrupting one of the criteria necessary for an attack by using lock-free data structures, universal hashing, balanced trees, randomized data structures, or partitioning of data structures. However, none of these solutions are competent in addressing the synchronization and framing attacks as these solutions are already vulnerable to attacks, introduce performance overhead, or require rewriting the kernel. Therefore, a new approach is necessary that is lightweight, can trigger an automatic response and recovery, and flexible to support multiple data structures.

To address the problem of adversarial synchronization, we design Trātṛ– a Linux extension to detect and mitigate synchronization and framing attacks with low overhead, prevent attacks from worsening, and recover by repairing data structures to their pre-attack performance. As the Linux kernel comprises multiple data structures, Trātṛ provides a general framework for addressing these attacks.

Trātṛ employs four mechanisms – tracking, detection, prevention, and recovery to detect and mitigate the attacks. Trātṛ tracks the data structure usage by each tenant and stamps each kernel object with the tenant information that it can use at a later point in detecting an attack and perform recovery. The detection mechanism periodically monitors synchronization stalls to detect an attack. Upon detecting a possible attack, Trātṛ uses the tracking information to identify the attacker and initiate mitigation mechanisms.

Upon detecting an attack, Trātṛ prevents the attacker from allocating more kernel objects, thereby preventing the attacker from further expanding the data structure. Prevention alone is not enough as victims of framing attacks can continue to observe poor performance. Therefore, Trātṛ initiates recovery mechanism to repair the data structure to the pre-attack

state. As the data structures are different, Trātṛ uses two different recovery methods - isolating the attacker or evicting the attacker's entries.

The current implementation of Trātṛ supports three different data structures - the inode cache, the futex table, and the directory cache tracking four different slab-caches associated with the three data structures and monitors two different synchronization mechanisms – spinlock and RCU.

We demonstrate the effectiveness of Trātṛ on the inode cache attack, the futex table attack, and the directory cache attack. We show that Trātṛ can quickly detect an attack, thereby preventing the victim from observing poor performance. Trātṛ can quickly initiate recovery to bring the victims' performance to pre-attack level. At a steady state, without an attack, Trātṛ imposes around 0-4% of tracking overhead on a variety of applications. The other three mechanisms incur less than 1.5% impact on the performance without an attack. We also show that Trātṛ can detect multiple attacks simultaneously, and it is easy to add a new data structure to Trātṛ.

1.4 Contributions

We list the main contributions of this dissertation.

- **Lock usage.** We present a new dimension to the existing properties of locks by introducing lock usage. Lock usage deals with the amount of time spent in the critical section while holding the lock. When multiple threads belonging to the same application or different applications participate in the lock acquisition process, lock usage plays a crucial role in ensuring all the participating threads can acquire the lock and execute critical sections.

- **Lock usage fairness.** We introduce a new factor to the existing fairness property of locks by coining lock usage fairness. Lock usage fairness guarantees that each competing entity receives a time window to use the lock perhaps once or many times. We call this window of opportunity lock opportunity and use it to measure lock usage fairness. Lock usage fairness becomes an important property in shared environments such as servers, operating systems, and hypervisors where multiple tenants can compete to acquire the shared data structures protected by synchronization mechanisms. Unfair lock usage can lead to performance and security issues.

- **Scheduler subversion.** For benign settings within shared environments, multiple tenants can introduce lock contention and also grow the data structures while executing their workloads. In doing so, the critical section size changes. When there is enough lock contention and the presence of variable-sized critical sections, there arises a new problem of scheduler subversion. Scheduler subversion is an imbalance created due to unfair lock usage where the lock usage determines the CPU share each thread obtains instead of the CPU scheduler.

- **Scheduler-Cooperative Locks.** To address the problem of scheduler subversion, we design Scheduler-Cooperative Locks, a new family of locks that can ensure lock usage fairness by aligning with the CPU scheduling goals. SCLs provide an equal or proportional lock opportunity to each thread within which each thread can acquire the lock one or more times.

  SCLs utilize three important techniques to achieve their goals: tracking lock usage, penalizing threads that use locks excessively, and guaranteeing exclusive lock access with lock slices. Using these three techniques – we implement three different SCLs: a user-level

mutex lock (u-SCL), a reader-writer lock (RW-SCL), and a simplified Linux kernel implementation (k-SCL).

- **Adversarial synchronization.** For hostile settings within shared environments, malicious actors can deliberately grow the shared data structures, thereby leading to artificial lock contention. When such a situation arises, there can be security arising leading to poor performance and denial-of-services.

  Using the idea of adversarial synchronization, we present a new class of attacks – synchronization attacks – that exploit Linux kernel synchronization to harm application performance. Using containers, we show how an unprivileged malicious actor can control the duration of kernel critical sections to stall victims running in other containers on the same operating system.

  A subset of the synchronization attacks termed framing attacks, not only stall the victims to access the synchronization mechanisms, but also forces the victims to execute the expanded data structures making their critical section also longer.

- **Trāṭr.** To address the problem of adversarial synchronization, we design Trāṭr, a Linux extension, to quickly detect synchronization and framing attacks, prevent attacks from worsening, and recover by repairing data structures to their pre-attack performance. Trāṭr relies on four mechanisms - tracking, detection, prevention, and recovery for efficient working. Trāṭr provides a general framework for addressing the attacks, and adding new data structures is easy.

1.5 Overview

We briefly describe the contents of the different chapters in the dissertation.

- **Background.** Chapter 2 provides background on concurrency synchronization and mutual exclusion. We discuss the properties of locks and how locks are designed. Finally, we discuss the implementation of commonly used synchronization primitives that we will use in the dissertation.

- **Lock usage fairness.** Chapter 3 introduces the idea of lock usage fairness and the importance of lock usage fairness in shared environments. We introduce the problem of scheduler subversion where the lock usage pattern dictates the CPU allocation instead of the CPU scheduler determining the share of the CPU each thread obtains. Further, we introduce the problem of adversarial synchronization by describing synchronization and framing attacks. We also illustrate synchronization and framing attacks on three Linux kernel data structures and explain the performance and economic impacts on the victims.

- **Scheduler-Cooperative Locks.** In Chapter 4, we start by studying how existing locks can lead to the scheduler subversion problem. Then, we introduce the idea of lock opportunity – a new metric to measure lock usage fairness. We discuss how lock usage fairness can be guaranteed by Scheduler-Cooperative Locks (SCLs), a new family of locking primitives that controls lock usage and thus aligns with system-wide scheduling goals. Lastly, we present the design, implementation, and evaluation of three different SCLs: a user-level mutex lock (u-SCL), a reader-writer lock (RW-SCL), and a simplified kernel implementation (k-SCL).

- **Taming adversarial synchronization attacks using Trātṛ.** In Chapter 5, we first start by discussing how existing solutions cannot solve the problem of adversarial synchronization. Then, we discuss the design, implementation, and evaluation of Trātṛ, a Linux kernel ex-

tension, to detect and mitigate synchronization and framing attacks with low overhead, prevent attacks from worsening, and recover by repairing data structures to their pre-attack performance.

- **Related Work.** In Chapter 6, we discuss how other research work and systems are related to this dissertation. We first discuss the work related to lock usage fairness, scheduler subversion, and adversarial synchronization. Then, we compare and contrast the work on SCLs and Trātṛ with existing work.

- **Conclusions and Future Work.** Chapter 7 summarizes this dissertation where we present a brief overview of lock usage and how the problems related to lock usage can be solved. We then present some lessons learned during the course of this dissertation, followed by the possible directions this work can be extended.

2 Background

In this chapter, we provide a background on various topics that are relevant to this dissertation. We start with a brief overview of concurrency, synchronization, and mutual exclusion in Section 2.1. Then, we discuss crucial lock properties, the five lock algorithm categories, and the waiting policy used to design locks in Section 2.2. Finally, we discuss the implementation of common synchronization primitives in Section 2.3.

2.1 Concurrency, Synchronization & Mutual Exclusion

Processor clock speeds are no longer increasing. Instead, computer architecture designers are focusing on designing chips with more cores. Therefore, to utilize these chips with more cores and achieve efficiency, system designers have to figure out how to decompose the programs into small parts that can run concurrently. These concurrent parts need to execute either in an orderly fashion or run independently and may interact with each other. Concurrent systems are systems that can be decomposed into various units of computation that can run at the same time. *Concurrency* is a property of such concurrent systems where multiple events in the form of computations are happening at the same time. As these events

can interact, the number of possible interleavings in the system can be extremely large, leading to complexity.

Whenever concurrent events happen, there will be various interleavings that are not desired, i.e., they are incorrect from the behavioral perspective and may lead to unforeseen problems. Such conditions are called as *race conditions*, and there has to be a way to avoid such undesired interleavings that can lead to non-deterministic results. *Synchronization* is an act of handling concurrent events in such a manner that we can avoid incorrect interleavings. Synchronization does so by making the concurrent events happen in certain time order.

A *synchronization primitive* is a mechanism that will ensure a temporal ordering of the events that we believe will always be correct. From a process synchronization perspective, threads and processes both have to rely on synchronization for correctness purposes. We use process and threads interchangeably to discuss the idea of process synchronization. Furthermore, when multiple processes use a synchronization primitive, they interact with the primitive itself to identify the ordering and coordinate to avoid undesired interleavings. Thus, we believe that using the synchronization primitive can influence the behavior of the process itself.

Every type of synchronization primitive offers either atomicity, ordering, or both. Atomicity ensures that a specified event or a sequence of events will either happen all together or not at all. Thus, there can never be any undesirable interleavings possible with atomicity. On the other hand, many times, just ensuring atomicity alone is not enough. An additional requirement of guaranteeing order is necessary too. Thus, ordering ensures that a specified event or a sequence of events never happens until a certain pre-condition is true.

One of the ways to implement atomicity is by enforcing *mutual exclusion*. One or more operations can be executed at a time by enforcing mutual exclusion. The set of operations that needs to execute in mutual ex-

clusion is called a *critical section*. The critical section accesses the shared variable, counters, or shared data structure, etc. Thus, mutual exclusion is a way to ensure that no two processes can exist in the critical section simultaneously. Multiple threads or processes that want to access the critical section will have to use a synchronization primitive to ensure mutual exclusion.

There are different ways to ensure mutual exclusion, such as using atomic instructions, locks, reader-writer locks, semaphores, monitors, and read-copy-update [138]. Locks are one of the widely-used synchronization primitives to build concurrent systems such as operating systems, user-space applications, and servers. A lock can be used across two or more processes or within a single process to ensure mutual exclusion across threads. Locks provide an intuitive abstraction of isolation where only one process is executing the critical section. When a lock is held, no other process is allowed to enter the critical sections that accesses the same shared data until the process completes executing the critical section and releases the lock.

There are several other synchronization primitives available, such as condition variables and monitors that developers can use to ensure ordering. Condition variable, a popular primitive, is a queue of waiting processes where the processes wait for a certain condition to be true. A condition variable is always associated with a mutual exclusion lock and can also be implemented by using other synchronization primitives such as semaphores [25].

2.2 Locks

We now discuss how locks can be designed. Firstly, we describe the various properties of locks. Then, we discuss the five locking algorithm categories depending on how the locks transfer the lock ownership, whether

they can address the performance demands, etc. Lastly, we discuss the waiting policy that various locking algorithms use to decide when the lock is unavailable.

To interact with the locks, processes and threads use the two most common APIs available with the locks – lock() and unlock(). The *lock*() or *acquire*() API is used to acquire the lock. Once the thread acquires the lock, it can continue with the execution of the critical section. If the lock is not free, the thread will wait until another thread completes the execution of the critical section. *Lock contention* happens when a thread tries to acquire a lock that is held by another process. The *unlock*() or *release*() API is used by the thread holding the lock to release the lock. Once the lock is released, another waiting thread can acquire the lock.

2.2.1 Crucial Lock Properties

Locks should exhibit certain properties so that one can evaluate the efficacy of the lock. The few crucial properties that lock designers should focus on are:

1. **Safety.** The safety property ensures that nothing bad will happen in the system. There are two aspects related to nothing bad. The first one is that the atomicity has to be guaranteed. The lock should always ensure mutual exclusion so that no two threads can execute the critical section simultaneously and lead to incorrect behavior.

   Secondly, locks should avoid deadlocks and is generally done by the user of the locks. When deadlocks happen, threads continue to wait for each other to release the locks, thereby getting into a bad state where no progress is being made. Locks that allow nesting or where the lock acquisition order is not maintained can lead to deadlocks. Many locks try to avoid deadlocks by breaking at least one of the

four conditions needed to end up in a deadlock that Coffman et al. identified [42].

2. **Liveness.** The liveness property ensures that eventually something good will happen, i.e., if the lock is free and a few threads are trying to acquire the lock, at least one thread will eventually acquire it and make forward progress.

   A stronger variant of liveness is starvation freedom; any given thread can never be prevented from making forward progress. A weaker variant, livelock freedom, deals with the fact that, as a whole, the system may make forward progress even though there may be threads that are starving. Deadlock avoidance approaches may lead to a livelock occasionally. A livelock is similar to a deadlock, where the processes do not make forward progress; however, the states of the processes involved in the livelock constantly changes with regard to one another [21].

3. **Fairness.** On closer inspection, livelock freedom does not adhere to the notion of fairness as there is a possibility that one or more threads may starve and eventually never make forward progress. Similarly, for starvation freedom, there is no bounded time associated with waiting. An arbitrary amount of time can be spent before the thread is guaranteed forward progress. Therefore, there arises a need to have some bounds on how long the thread should wait before it is guaranteed to make forward progress.

   By bounding the wait times to acquire the locks, each thread will get a fair chance to acquire the lock without starving. The bounding time can be defined in any way possible. Generally, the bounding wait time is defined by the order in which the threads acquire the lock. Thus, no thread can access a lock twice while some other

threads are kept waiting. We call this type of fairness *acquisition fairness*.

To guarantee lock acquisition fairness, the locks need to maintain some state and remember the order in which the threads tried to acquire the lock. This state information can be either simple or complex depending on the assumptions.

4. **Performance.** The performance of the lock largely depends on two aspects. The first aspect is overhead, and the second aspect is scalability. Ideally, a good locking algorithm should have low overhead so that frequent usage of the lock is possible and should scale well such that its performance should not degrade as the number of CPUs increases.

   Lock overhead deals with various factors such as the memory space allocated for the locks, initializing and destroying the locks, and the time taken to acquire and release the locks with and without lock contention. As the locks need to maintain states, there is always some memory allocation associated with the locks. For simple locks, the memory space needed may be a few bytes. While for complex locks, as more information needs to be stored, the memory space needed is high. As thousands of locks may be used to implement concurrent systems, the total memory space needed becomes crucial [55].

   Similarly, there may be use-cases where the locks need to be initiated and destroyed frequently. The lock design should keep this factor in mind while designing the locks. There can be performance issues if the lock initialization and destroying takes a significant amount of time.

   The time taken to acquire and release the locks should be minimal. The performance of the lock should not vary much irrespective of

how many threads are participating in the lock acquisition process. If the threads spend enough time performing locking operations, the performance of the application will be impacted. Lock designers should keep in mind different cases that are worth considering, such as what is the overhead when only a single thread is acquiring and releasing the lock; what happens when there are multiple threads participating in the lock acquisition process, i.e., during heavy lock contention.

There is a significant relationship between the cache-coherence protocols of the underlying hardware and the locks [50]. As the locks are implemented using atomic instructions, their performance depends on the cost associated with these atomic instructions. Execution of these atomic instructions does not scale well in the presence of non-uniform memory access (NUMA) nodes. More so, with contention, the performance of the atomic instructions degrade quickly, leading to a performance collapse. Hence, while designing the locks, one will have to keep in mind the presence of NUMA nodes and the number of CPUs in the system.

Locks should enable correctness while still ensuring that they incur low-overhead and can scale well with the number of CPUs. Researchers and practitioners have put forth decades of effort to design, implement, and evaluate various synchronization schemes [10, 16, 26, 28, 37, 38, 45, 48, 50, 53, 56, 57, 62, 64, 73, 74, 85, 86, 101, 102, 109, 119, 131, 133, 135, 164].

Generally, simple locks are efficient in terms of overhead as they do not maintain too much state information leading to low memory overhead. However, they do not scale well as the number of CPUs increase. Furthermore, they cannot guarantee acquisition fairness as there is not enough state information available to decide on lock ownership. On the

other hand, complex locks will have memory overhead. Still, they may guarantee acquisition fairness as they have enough state information to identify the lock ownership.

Even though we do not discuss the properties of other synchronization primitives, the properties that we discuss for the lock also hold for other synchronization primitives. We will now discuss the simple and complex locks and the properties they guarantee.

2.2.2 Categorizing Lock Algorithms

The body of existing work on locking algorithms is rich and diverse. Numerous optimized locking algorithms have been designed over the past few decades to ensure mutual exclusion and address the concerns related to scalability, low overhead, and fairness. All of these locking algorithms can be split into five categories – competitive succession, direct handoff succession, hierarchical approaches, load-control approaches, and delegation-based approaches [71].

The first two categories are based on how the lock ownership is transferred from the current lock owner to the next owner while releasing the lock. The other three categories support NUMA machines by building hierarchies to reduce lock migration across NUMA nodes, controlling how many threads participate in the lock acquisition process, and delegating the execution of the critical sections to the current lock owner. While the last three categories focus more on improving lock performance, many locks in these categories guarantee acquisition fairness. Compared to the first two categories that have a simple design, the locks belonging to the last three categories have a complex design.

1. **Competitive succession.** Locking algorithms in this category are one of the simplest algorithms that comprise of one or a few shared variables accessed via atomic instructions. There is no specific di-

rection on passing the lock ownership from the current lock owner to other competing threads. Once the current lock owner releases the lock, all the waiting threads compete to be the next lock owner.

As these algorithms use atomic instructions, the cache-coherence protocol of the underlying hardware plays a role in deciding who the next lock owner will be. Due to this reason, any thread can acquire the lock in any order leading to unfair behavior and starvation. There can even be instances when an arriving thread may acquire the lock while the existing waiting threads continue to wait to acquire the lock [53]. Multiple threads concurrently access the shared variables using atomic instructions generating cache-coherence traffic leading to performance problems.

Simple spinlock [138], backoff spinlock [16, 109], test and test-and-set lock [16], mutexee lock [61], and the standard Pthread mutex lock [92] are the examples of locks that belong to this category.

2. **Direct handoff succession.** Locking algorithms in this category always decide on who the next owner is going to be instead of allowing all the competing threads to compete for the lock ownership. The threads participating in the lock acquisition process can spin on a local variable, reducing the cache-coherence traffic. Hence, such algorithms are scalable and incur low overhead as the waiting threads do not have to access the shared variables for lock ownership.

   MCS [109, 138], CLH [45, 102, 138], ticket lock [21, 138], and partitioned ticket lock [48] are examples that follow the strategy of directly handing off the lock ownership to the chosen one. Even though ticket lock and partitioned ticket lock uses a global variable to keep note of the successor, they can be implemented efficiently to reduce cache-coherence traffic.

3. **Hierarchical approaches.** Locking algorithms in this category are designed to scale well on NUMA nodes. The algorithms do so by ensuring that the lock ownership does not migrate across the NUMA nodes frequently. Rather, the lock ownership transfers between different NUMA nodes periodically where all the threads running on that NUMA node can acquire the lock. These locking algorithms are built using either competitive succession or direct handoff succession. HBO [131], HCLH [141], FC-MCS [56], HMCS [37] are examples of the locks that belong to this category.

   Locks based on the lock cohorting framework [57] also belong to this category. A cohort lock combines two locking algorithms where one lock is used as a global lock to transfer the lock ownership at the NUMA node level. In contrast, the other lock is used locally to transfer lock ownership to participating threads within the NUMA node. For example, the CBO-MCS lock corresponds to the global lock being backoff spinlock and the local lock being MCS lock. Similarly, the C-PTL-TKT lock corresponds to the global and local lock as partitioned ticket lock and ticket lock, respectively. C-TKT-TKT (HTicket) [50] lock comprises both the global and local lock being ticket lock.

4. **Load-control approaches.** Locking algorithms that prevent performance collapse by limiting the number of participating threads come under the load-control approaches. These locks can adapt to the changing lock contention and accordingly decide on whether to allow multiple threads to participate in the lock acquisition process or not. Locks such as MCS-TP [73], GLS [18], SANL [164], LC [82], AHMCS [38], and, Malthusian algorithms [53] fall under this category.

5. **Delegation-based approaches.** Delegation-based approaches com-

prise of threads delegating the execution of the critical section to another thread typically holding the lock. This way, threads can avoid the transfer of lock ownership, and only one thread can execute the critical section on behalf of others. Doing so helps in improving the performance as there is less data migration across caches, improving the cache locality. Locks such as Oyama [119], Flat Combining [74], RCL [99], FFWD [135], CC-Synch [62], and DSM-Synch [62] fall under this category.

2.2.3 Waiting Policy

Apart from deciding who the next lock owner will be, the locking algorithms also have to decide what the waiting threads should do when the lock is not readily available. The locking algorithms use three main waiting policies.

1. **Spinning.** This is one of the simplest approaches where the waiting threads loop continuously to check the lock status and acquire the lock once it is available. As the threads continuously probe the lock status without doing any useful work, in this approach, other threads that are waiting for the CPU are not scheduled, thereby wasting CPU. Additionally, such an approach might also not work where saving power is crucial.

   Modern processors provide special instructions such as PAUSE to inform the CPU that the thread is spinning, and the CPU can release shared pipeline resources to the sibling hyperthreads [53]. Techniques such as fixed or randomized backoff are also used to reduce the cache-coherence traffic. Modern hardware also provides facilities such as lowering the frequency of the waiting thread's core [160] or notifying the core to switch to idle state [61]. Often, the lock developers rely on the sched_yield() system call to volun-

tarily relinquish the CPU and let other threads be scheduled [21]. However, when there are very few threads waiting to be scheduled, yielding the CPU is not guaranteed to be useful, and the waiting thread will often trigger context switches degrading the performance.

2. **Blocking or immediate parking.** With the blocking approach, a waiting thread immediately blocks until the thread is again able to acquire the lock. Whenever the thread needs to block, the CPU scheduler needs to be informed same to not schedule the thread until the lock is available. On Linux, this is often done by using the futex system call [63]. While releasing the lock, the lock holder can inform the CPU scheduler to allow the scheduling of the blocking thread. The futex system call allows the user to specify how many threads should be woken to participate in the lock acquisition process.

   Unlike the spinning approach, this approach is often efficient when the total number of threads exceeds the available CPUs. As the waiting threads immediately block, other threads do get a chance to use the CPU for other work. If all the threads wait to acquire the lock, only the thread holding the lock will run.

3. **Spin-then-park.** The spin-then-park approach takes the middle ground between the spinning and block approaches. It is a hybrid approach where a fixed or adaptive spinning threshold can be used to first spin until a certain threshold and then can block the thread if the lock is still not available [84]. This approach is useful as it tries to avoid the high cost of context switching the thread while blocking but at the same time avoids the threads from spinning for a long time, allowing other threads to schedule.

Generally, deciding the waiting policy approach is not directly related to the five locking algorithms categorization we discussed above. One can choose any waiting policy which designing the locks. However, there are few things developers have to keep things in mind. If the critical section is small, using the blocking approach will not be efficient as there is a high cost associated with the context switching. For such critical sections, the spinning approach is beneficial as all the threads will acquire the lock in a short period. Conversely, if the critical section size is long, blocking the threads immediately is a good idea to avoid the threads spinning unnecessarily.

2.3 Common Synchronization Primitives Implementation

Now that we have discussed how locks can be designed, we will discuss the design of a few synchronization primitives that we will use in this dissertation. We start by discussing a few common locks used to design user-space applications and the Linux kernel. Then, we will discuss the implementation of two other types of synchronization primitives – reader-writer locks and read-copy-update (RCU).

2.3.1 Pthread Spinlock

The pthread spinlock is one of the simplest and most widely-used spinlocks provided by the pthread library [88]. We show the pseudo-code of the pthread spinlock in Listing 2.1. The lock comprises only a single shared integer variable, and it uses atomic instructions for locking and unlocking purposes. The lock is initialized to free so that any arriving thread can acquire the lock.

```
typedef volatile int pthread_spinlock_t;

int pthread_spin_init(pthread_spinlock_t *lock)
{
  // Initialize the lock to 0 (free).
  int init_value = 0;
  __atomic_store(lock, &init_value, __ATOMIC_RELAXED);
  return 0;
}

int pthread_spin_lock(pthread_spinlock_t *lock)
{
  int expected = 0;
  if (__atomic_compare_exchange_n(lock, &expected, 1, 1, __ATOMIC_ACQUIRE,
                                  __ATOMIC_RELAXED)) {
    // Acquired the lock, return now.
    return 0;
  }
  // Spin until the lock is not free.
  do {
    // First check if the lock is free.
    do {
      expected = __atomic_load_n(lock, __ATOMIC_RELAXED);
    } while(expected != 0);
  } while(!__atomic_compare_exchange_n(lock, &expected, 1, 1, __ATOMIC_ACQUIRE,
                                       __ATOMIC_RELAXED));
  // Finally, the lock is acquired.
  return 0;
}

int pthread_spin_unlock(pthread_spinlock_t *lock)
{
  int unlock = 0;
  // Mark the lock free.
  __atomic_store(lock, &unlock, __ATOMIC_RELEASE);
  return 0;
}
```

Listing 2.1: Pthread spinlock pseudo-code

While trying to acquire the lock, the thread first uses an atomic instruction to set the shared variable value to 1. If it succeeds, it returns and starts executing the critical section. If it fails, then it spins until the lock is free. While spinning, the thread tries first to check the value of the shared variable. Then, the thread again tries to atomically set the shared variable value to 1 to acquire the lock on finding it free. If it fails, then it again loops.

One can observe that the memory overhead of the pthread spinlock is minimal while the implementation is also extremely simple. However, pthread spinlock does not guarantee lock acquisition fairness and a thread, either a spinning thread or newly arriving thread can succeed while executing the atomic instruction.

2.3.2 Pthread Mutex

The pthread mutex is another widely used locking primitive provided by the pthread library. Unlike the spinlock version, the mutex lock blocks if the lock is not free instead of spin-waiting. The pseudo-code of pthread mutex is shown in Listing 2.2. In addition, one can configure the pthread mutex with the PTHREAD_MUTEX_ADAPTIVE_NP initialization attribute to let the thread spin on the lock until either the maximum spin count (default is 100) is reached, or the lock is acquired [91].

For a spinlock, the lock can either be in 0 (free) or 1 (acquired) state. For a mutex lock, the lock can be in one of three states – 0 for free, 1 for acquired with no waiters, i.e., no other thread is blocked or about to block, and >1 for acquired with waiters present.

When a thread tries to acquire the lock, it checks if the lock is free or not. If it is free, the lock value is set to 1 atomically to indicate that it is acquired. If the lock is already acquired, the thread will block itself until it acquires the lock and sets the lock value to 2. The futex syscall is used to block the threads.

```
int sys_futex(void *addr1, int op, int val1, struct timespec *timeout,
              void *addr2, int val3)
{
  return syscall(SYS_futex, addr1, op, val1, timeout, addr2, val3);
}

typedef int mutex;

int mutex_init(mutex *m)
{
  *m = 0; // Init the lock to 0.
  return 0;
}

int mutex_lock(mutex *m)
{
  int c = 0;
  __atomic_compare_exchange_n(m, &c, 1, 0, __ATOMIC_SEQ_ACQUIRE,
                              __ATOMIC_RELAXED);
  if (!c) {
    return 0; // Got the lock, return now.
  } else if (c == 1) {
    /* The lock is now contended. */
    c = __atomic_exchange_n(m, 2, __ATOMIC_SEQ_CST);
  }

  while (c) {
    /* Wait in the kernel. */
    sys_futex(m, FUTEX_WAIT_PRIVATE, 2, NULL, NULL, 0);
    c = __atomic_exchange_n(m, 2, __ATOMIC_SEQ_CST);
  }
  return 0;
}

int mutex_unlock(mutex *m)
{
  int state = __atomic_exchange_n(m, 0, __ATOMIC_SEQ_CST);

  if (state > 1) {
    /* We need to wake someone up. */
    sys_futex(m, FUTEX_WAKE_PRIVATE, 1, NULL, NULL, 0);
  }
  return 0;
}
```

Listing 2.2: Pthread mutex pseudo-code

While releasing the lock, the thread will first unconditionally set the lock to 0. Then it will wake up one thread who will again try to acquire the lock by issuing the futex syscall. If it succeeds, then it will return else again block itself.

There are two aspects to note here. Firstly, as the waiting threads immediately block, the waiting threads will not use the CPU, making it an efficient lock. Secondly, by frequently calling futex syscall to block, a lot of time is spent moving from user-mode to kernel-mode and back.

2.3.3 Ticket Lock

The pthread Spinlock and pthread Mutex are examples of competitive succession category where the lock ownership order is not predetermined. The threads are free to acquire the lock in any order. Such a design can lead to a scenario where one thread waits for a very long time to acquire the lock and starve, leading to poor performance.

The ticket lock is an example of the direct handoff category and addresses starvation [139]. The ticket lock assigns the lock ownership depending on the order in which the threads try to acquire the lock, i.e., in the first-in-first-out order. All the threads that requested to acquire the lock before the arriving thread are guaranteed to acquire the lock before the arriving thread acquires the lock.

By using the atomic instruction fetch_add, one can implement a ticket lock. A simple implementation of the ticket lock is shown in Listing 2.3. Whenever an arriving thread tries to acquire the ticket lock, it executes the atomic instruction to obtain a ticket number. The arriving thread then waits for its turn by continuously checking if its ticket number is the same as the currently serving ticket. In that case, the thread acquires the ticket lock and starts executing the critical section. While releasing the lock, the lock owner simply increases the currently serving ticket by 1 to let the next thread acquire the lock.

```
typedef struct lock {
  int next_ticket;
  int currently_serving;
} ticket_lock;

void ticket_lock_init(ticket_lock *lock)
{
  lock->next_ticket = 0;
  lock->currently_serving = 0;
}

void ticket_lock_acquire(ticket_lock *lock)
{
  // Obtain my ticket.
  int my_ticket = __atomic_fetch_add(&lock->next_ticket, 1, __ATOMIC_SEQ_CST);
  // Loop until it is my turn to acquire the lock.
  while (lock->currently_serving != my_ticket);
}

void ticket_lock_release(ticket_lock *lock)
{
  // Move the current ticket by 1.
  lock->currently_serving = lock->currently_serving + 1;
}
```

Listing 2.3: Ticket lock pseudo-code

Even though the code we have mentioned here simply spins until its turn comes, a lock designer can employ either a fixed or randomized back-off strategy, yield the CPU, or block the thread if the critical section size is known and the time to acquire the lock is more than the context-switch time.

2.3.4 MCS & K42 variant

Like ticket lock, MCS lock also belongs to the direct handoff category. Threads can perform an arbitrary number of remote accesses leading to cache-coherence traffic while trying to acquire the ticket lock. MCS lock solves this problem by ensuring that every thread spins on a local variable that lies in the stack frame of the thread [138]. The pseudo-code for the MCS lock is shown in Listing 2.4.

Each thread allocates a node containing a queue link and a locked vari-

```
struct lnode {
  struct lnode *next; // Use for queueing.
  int locked; // Local variable for spinning.
};

typedef struct lock {
  struct lnode *tail;
} mcs_lock;

void mcs_acquire (mcs_lock *lock, struct lnode *p)
{
  struct lnode *prev;
  p->next = NULL; // Init the new node.
  p->locked = 0;
  // Attach the thread to the end of the tail.
  __atomic_exchange(&lock->tail, &p, &prev, __ATOMIC_SEQ_CST);

  if (prev != NULL) {
    prev->next = p;
    while (!p->locked ); // Spin until its your turn.
  }
}

void mcs_release(mcs_lock *lock, struct lnode *p)
{
  struct lnode *next;
  struct lnode *desired;
  struct lnode *cmp_pointer;
  // Get the next rightful owner.
  __atomic_load(&p->next, &next, __ATOMIC_SEQ_CST);

  if (next == NULL) { // Set tail to NULL.
    desired = NULL;
    cmp_pointer = p;
    if (__atomic_compare_exchange(&lock->tail, &cmp_pointer, &desired, 1,
                                  __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)) {
      return;
    }
    do { // Some other thread just arrived.
      __atomic_load(&p->next, &next, __ATOMIC_SEQ_CST);
    } while (next == NULL);
  }
  next->locked = 1;
}
```

Listing 2.4: MCS lock pseudo-code

able. The threads use this locked variable to spin until it's time for the thread to acquire the lock. A thread that either holds the lock or waits for the lock form a chain together in first-in-first-out order. The lock ownership transfers from one node to another node that forms the queue. As shown in the listing, the lock itself is a pointer to the node of the thread at the tail of the queue. The lock is free if the tail pointer is null.

While trying to acquire the lock, the thread first allocates a node, initializes the node fields, and then swaps it to the end of the queue's tail. If the swap value is NULL, the thread has acquired the lock. If the swap value is non-NULL, the thread will spin wait by looking at the node's locked variable.

While releasing the lock, the thread reads its node's next pointer to identify the next lock owner. Then the thread changes the value of the locked variable of the successor thread to indicate the successor thread that it can now acquire the lock. If the thread finds that there is no successor, the thread sets the tail pointer to NULL.

As each thread spins on a separate location, there is less cache-coherence traffic generated, leading to better performance. However, to acquire and release the MCS lock, the node and the lock need to be passed to the acquire and release API. We now show a variant of the MCS lock called the K42 variant, where there is no need to pass on the node to acquire and release API [138].

Unlike the MCS lock, the K42 variant has the tail and next pointers in the node structure. The pseudo-code for the K42 variant is shown in Listing 2.5. When the lock is free, the tail and the next pointers are set to null. Whenever an arriving thread tries to acquire the lock, it replaces the null tail pointer with a pointer to the lock itself to notify that no other thread is waiting.

If a second thread arrives, it will make the lock's tail and next pointer point to the node of the second thread. Then it spins to wait for the lock

```
struct lnode {
  struct lnode *next;
  struct lnode *tail;
};

struct lnode *waiting = 1;

typedef struct lock {
  struct lnode queue;
} k42_lock;

void acquire (k42_lock *lock)
{
  struct lnode *prev, *desired, *new, *succ, *temp, *oldnew;

  while (1) {
    desired = NULL;
    prev = lock->queue.tail;

    if (prev == NULL) { //Lock appears to be free.
      temp = &lock->queue;
      if (__atomic_compare_exchange(&lock->queue.tail, &desired, &temp, 1,
                                    __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)) {
        return;
      }
    } else {
      new = (struct lnode*)malloc(sizeof(struct lnode));
      new->next = NULL;
      new->tail = waiting;

      // We are in line now.
      if (__atomic_compare_exchange(&lock->queue.tail, &prev, &new, 1,
                                    __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)) {
        prev->next = new;
        while (new->tail == waiting); // Spin. Wait for your turn.
        succ = new->next;

        if (succ == NULL) {
          lock->queue.next = NULL;
          temp = &lock->queue;
          oldnew = new;

          // Try to make lock point to itself.
          if (!__atomic_compare_exchange(&lock->queue.tail, &oldnew, &temp, 1,
                                         __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)) {
            // Someone else got in between. Keep trying.
```

```
            do {
              succ = new->next;
            } while (succ == NULL);

            lock->queue.next = succ;
          }

          return;
        } else {
          lock->queue.next = succ;
          return;
        }
      }
    }
  }
}

void release (k42_lock *lock)
{
  struct lnode *succ;
  struct lnode *desired = NULL;
  struct lnode *temp;

  succ = lock->queue.next;

  if (succ == NULL) {
    temp = &lock->queue;

    // Inform the next owner about the lock release.
    if (__atomic_compare_exchange(&lock->queue.tail, &temp, &desired, 1,
                                  __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST)) {
      return;
    }

    do {
      succ = lock->queue.next;
    } while (succ == NULL);
  }

  // Setting lock free. No waiter is present.
  succ->tail = NULL;
  return;
}
```

Listing 2.5: K42 variant pseudo-code

owner to free the lock. While waiting, if a third thread arrives, it will form a queue by updating the next pointer of the second thread, and the lock's tail pointer points to the third thread.

When the lock is released, the lock owner will find the next-in-line thread's node and change its node's tail pointer value to null, allowing the spinning thread to proceed. However, before entering the critical section, it will have to move the pointers appropriately to the next thread in the queue.

As one may have noticed, the acquire and release APIs only need to pass in the k42_lock variable, unlike the MCS lock, where two parameters need to be passed. Thus, both the MCS lock and K42 variant will ensure lock acquisition fairness and not starve any thread.

2.3.5 Linux Kernel's Queued Spinlock

Queued spinlock is a locking mechanism in the Linux kernel, a special type of spinlocks that also guarantees lock acquisition fairness. This is enabled by default in the Linux kernel. The implementation of the Linux kernel is different than the implementation of pthread spinlock that we discussed earlier. As the name suggests, the queued spinlock is a combination of the spinlock and MCS lock, thereby guaranteeing lock acquisition fairness.

The current implementation of the queue spinlocks uses a multi-path approach. A fast path is implemented using the atomic instructions, and a slow path is implemented as an MCS lock when multiple threads are trying to acquire the lock. The lock is four bytes and is divided into three parts – the lock value, the pending bit, and the queue tail.

Preemption is disabled when the threads try to acquire the lock. As a thread will not be migrating to another CPU while trying to acquire the spinlock, only one CPU can contend for only one spinlock at any given point in time. Therefore, for each CPU, a set of queue nodes is pre-

allocated that can be used to form a waiting queue and can encode the queue address to fit the 4 bytes easily.

A thread acquiring a spinlock tries first to flip the value of the lock atomically from 0 to 1. If it succeeds, then it has the lock, and it can continue with the critical section execution. If it fails, it first checks if there is contention on the lock by checking if any other bit is set (either the pending bit or the queue tail).

When there is no contention, the thread tries to set the pending bit atomically, and then spin waits for a bounded time until the current lock owner released the lock. If there is contention or the lock is not released before the bounded time, the thread prepares to add itself to the queue.

After setting up the pre-allocated per-CPU queue node, it then adds itself to the queue and waits for itself to be at the head of the queue to be the lock owner. Like the MCS lock, the thread waits on the local variable, thereby reducing the cache-coherence traffic. Once at the head of the queue, the thread will wait for the current lock owner and another thread waiting on the pending bit to release the lock. The thread can do so by checking the lock's value and the pending bit to be 0. When this happens, the thread can non-atomically acquire the lock and inform its successor in the queue that it will be at the head of the queue now. While releasing the lock, the thread updates the lock value to 0.

2.3.6 Reader-Writer Lock

So far, we have seen the implementation of the mutual exclusion locks that always guarantees that only one thread is executing the critical section while all the other threads wait until the thread finishes with the critical section execution. With such a setting, it is not possible to allow multiple threads to execute the critical section even though they are not going to modify the shared state.

```
int WA_FLAG = 1;
int RC_INC = 2;
typedef volatile int rwlock;

// Initialize the lock.
void rwlock_init(rwlock *lock)
{
    int init_value = 0;
    __atomic_store(lock, &init_value, __ATOMIC_RELAXED);
}

void rwlock_reader_acquire(rwlock *lock)
{
  // Reader acquiring lock. Increment by 2.
  __atomic_add_fetch(lock, RC_INC, __ATOMIC_SEQ_CST);
  // Spin until writer is present.
  while ((__atomic_load_n(lock, __ATOMIC_SEQ_CST) & WA_FLAG) == 1);
}

void rwlock_reader_release(rwlock *lock)
{
  // Reader releasing lock. Decrement by 2.
  __atomic_sub_fetch(lock, RC_INC, __ATOMIC_SEQ_CST);
}

void rwlock_writer_acquire(rwlock *lock)
{
  int expected_writers = 0;
  // Spin until either readers or other writers present.
  while (!(__atomic_compare_exchange_n(lock, &expected_writers, WA_FLAG, 1,
                                       __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST))) {
    expected_writers = 0;
  }
}

void rwlock_writer_release(rwlock *lock)
{
  // Writer releasing lock. Decrement by 1.
  __atomic_sub_fetch(lock, WA_FLAG, __ATOMIC_SEQ_CST);
}
```

Listing 2.6: Centralized reader-preference reader-writer pseudo-code

Reader-writer locks relax this constraint of the mutual exclusion locks by allowing multiple threads to execute the critical section simultaneously when they do not modify the shared state. Such threads are termed readers as they are only reading the shared state. On the other hand, if a thread needs to modify the shared state, it should exclusively acquire the lock. Such a thread is termed as a writer. Thus, the readers and writers are mutually exclusive to each other; readers can always execute concurrently.

There are two modes of reader-writer locks – a reader preference lock and a writer preference lock. A reader preference lock allows a newly arriving reader to bypass an existing waiting writer and continue executing the critical section. There is a possibility that a writer may starve if readers continue to arrive before all the readers complete the execution of the critical section. A writer preference lock does not allow a newly arriving reader to bypass an existing waiting writer. Instead, the arriving reader will wait for the waiting writer to complete its execution. This way, the writers are not starved and always get a chance to make forward progress.

Reader-writer locks can be implemented using either centralized algorithms or a queue-based approach [138]. In this thesis, we only focus on the centralized algorithm and reader-preference reader-writer lock, and its implementation is shown in Listing 2.6. There are four APIs provided; the readers use rwlock_reader_acquire() and rwlock_reader_release() to acquire and release the lock, respectively. The other two API's The writer uses rwlock_writer_acquire() and rwlock_writer_release() to acquire and release the lock, respectively.

The lock is a shared variable that is accessed and updated using atomic instructions. When the value of the variable is 0, the lock is free. If the variable's value is even, there are one or readers currently executing the critical section. If the variable's value is 1, then there is a writer currently executing the critical section.

When a reader tries to acquire the lock, it increases the value of the variable by 2. It then spins until the variable value is even, i.e., the writer releases the reader-writer lock. If there is no writer present, the reader returns and starts executing the critical section. When the reader releases the lock, the value of the variable is decreased by 2.

When a writer tries to acquire the lock, it uses the atomic instruction to check if the lock's value is 0 and then sets the variable to 1. If there is any reader or a writer present, the variable's value will not be 0. After setting the variable value to 1, it can proceed with executing the critical section. At the time of releasing the lock, the writer decrements the value of the variable by 1.

2.3.7 Read-Copy Update

Read-Copy-Update or more commonly known as RCU is a type of synchronization mechanism originally developed for Linux kernel usage [108] and recently extended for user-space usage [52]. RCU is a library that allows multiple subsystems in the Linux kernel to access shared data structures efficiently without much overhead. RCU is designed for situations involving read-mostly workloads. There is close to zero overhead for the read synchronization, while for the updaters, especially the deleters, there is a very heavy overhead associated.

RCU comprises three mechanisms that combined helps in a very efficient synchronization mechanism [105]. The three mechanisms are used to handle the insertion, allowing multiple versions for readers to read data without any special synchronization effort and deletions.

The first mechanism that deals with insertions help with the ability to safely access or scan the data even though another thread is concurrently updating the data. RCU maintains multiple versions of the data while updating, and the second mechanism handles that aspect. Therefore, the readers will be either be accessing the older data or the new data.

```
void rcu_read_lock()
{
  disable_preemption();
  level[cpu_id()]++; // To handle nesting
}

void rcu_read_unlock()
{
  level[cpu_id()]++; // To handle nesting
  enable_preemption();
}

void synchronize_rcu()
{
  loop through all CPUs
        check whether the threads have context-switched
}
```

Listing 2.7: Read-Copy-Update (RCU) pseudo-code

The third mechanism deals with the deletion of the data. Any thread that wants to delete the data will have to ensure that all the existing readers drain before the data can be safely deleted. As the readers may be accessing the older or newer data, there is no way to identify if any thread is accessing the old data. Therefore, RCU delays the deletion of the old data until all the threads are done with reading work.

RCU does not provide any specific ability to handle concurrent insertions or deletions. Therefore, one must rely on other synchronization mechanisms such as locking to ensure mutual exclusion amongst concurrent updaters.

RCU provides three basic primitives that kernel developers can use. The first primitive deals with specifying the read-side critical sections. A reader enters the read-side critical section by calling rcu_read_lock() and exits the critical section by calling rcu_read_unlock(). The pseudo-code of both the APIs is shown in Listing 2.7. rcu_read_lock() simply disabled the preemption and rcu_read_unlock() enables preemption. RCU allows nesting within the read-side critical section.

The second primitive is associated with the RCU synchronization and

is used to handle the deletions. To ensure synchronization between deleting data and reading the data concurrently, a thread willing to delete data must call synchronize_rcu(). The call will only return when all the RCU read-side critical sections there were currently executing at the time of the synchronize_rcu() completes. This way, RCU guarantees that no reader is still accessing the older data, and hence it is safe to delete the data. Any new thread that wants to enter the read-side critical section can do so as RCU will ensure that it always accesses the newer data.

As preemption is disabled during the read-side critical section execution, synchronize_rcu() can check if all the CPUs have at least done one context switch or not. Context switch will only happen after the threads have called read_rcu_unlock(). Apart from the synchronous way of waiting for all the read-side critical sections to be over, RCU also provides an asynchronous way. Threads that do not want to wait can call call_rcu(). RCU will register a callback that a background thread will execute once all the readers are done executing the read-side critical section.

Lastly, RCU provides two architecture-specific APIs for readers and updaters to properly handle the memory orderings. Readers use rcu_deference() to fetch a pointer for dereferencing purposes. Updaters use rcu_assign_pointer() to modify the pointers within the critical section.

2.4 Summary

In this chapter, we presented the background essential to understanding this dissertation. We discussed various nuances of concurrency, synchronization, and mutual exclusion. Then, we discussed the lock properties, different locking algorithm categories, and the waiting policies used to design locks. Finally, we discussed the design and implementation of commonly used synchronization primi-

tives. The source code of the common locks can be accessed at https://research.cs.wisc.edu/adsl/Software/.

3
Lock Usage

Locks and their properties form an integral component of concurrent systems. Locks generally should have low overhead, enabling frequent usage without excessive overhead [50, 109]. Locks should also exhibit other properties, such as starvation-avoidance [109, 150]. One critical property is *fairness*. If two threads are competing for the same lock, and the lock is granted more frequently to one thread, the other will receive a smaller share of the desired lock, perhaps subverting system-wide goals.

Lock fairness is a well-studied problem; researchers have crafted many solutions that provide certain fairness properties quite effectively [45, 53, 73, 90, 102, 109, 139, 147, 150]. Consider a simple ticket lock [109]; by granting a ticket number to each interested party, the lock can ensure what we call *acquisition fairness*. In this case, under heavy competition, each party will be ensured of a fair chance to enter the critical section in question.

Consider a linked list supporting insert and find operations protected by a lock as shown in Listing 3.1. The insert operation is simple, where an entry is added at the head. On the other hand, the find operation has to traverse through the linked list to find if an entry exists or not. If one thread performs an insert operation, it will hold the lock for a shorter time than another thread that performs a find operation as it will have to traverse the entire list. Thus, the critical section size may vary for two different operations trying to access the same data structure.

```
struct node {
  int data;
  struct node *next;
};

void insert(struct node **list, struct node *n) {
  lock();
  n->next = *list;
  *list = n;
  unlock();
}

struct node *find(struct node **list, int data) {
  lock();
  struct node *n = *list;

  while (n) {
        if (n->data == data) {
          unlock();
          return n;
        }

        n = n->next;
  }

  unlock();
  return NULL;
}
```

Listing 3.1: Simple linked list example

The difference between the critical section sizes may vary depending on the state of the data structure. For example, the critical section size will be longer when the linked list has millions of entries than a linked list state with thousands of entries. As the data structure grows and shrinks depending on the workload, there will be a difference in the critical sec-

tion size. Thus, one can observe that the critical section size dynamically changes and is not a static property. In this chapter, we ask the question - what is the impact on performance and fairness when a single lock protects critical sections that are variable-sized?

We have found that a critical lock property, which we call *lock usage*, is missing from previous approaches. Consider two threads competing (repeatedly) for the same lock. If one thread stays within the critical section for a longer period and the other for a shorter period, the first thread will dominate lock usage. If one thread dwells longer in the critical section than other threads, the imbalance can lead to problems. Further expanding, consider the lock is a ticket lock in the scenario above. Despite guaranteeing lock acquisition fairness by alternating turns through the critical section, the thread that stays within the critical section longer receives more *usage* of the resource. Thus, we observe that the existing locks that guarantee lock acquisition fairness cannot address the lock usage imbalance created due to the variable-sized critical sections.

As the first thread in the scenario spends more time in the critical section, it will receive an equivalent CPU time. On the other hand, the second thread will receive lesser CPU time. Thus, although the CPU scheduling goals may have been to give each thread an equal share of CPU, the lock usage imbalance will end up subverting the scheduling goals. We call this new problem *scheduler subversion*. The scheduler subversion problem is similar to priority inversion, where a high priority task is indirectly preempted by a lower priority task, thereby inverting the priorities of the two tasks [140].

When scheduler subversion arises, the lock usage pattern dictates the share instead of the CPU scheduler determining the proportion of the processor each competing entity obtains. When threads regularly compete for the same lock, and there is the presence of variable-sized critical sections; there is a likelihood of having the scheduler subversion problem.

As mentioned earlier, the data structure state can impact the critical section size. Consider a thread that manages to expand a data structure intentionally with malicious intent. Two problems can arise due to the malicious activity of the thread.

First, after expanding the data structure, the malicious thread can deliberately access the expanded structure making the critical section even longer. As a result, other threads competing to access the lock will have to wait longer to acquire the lock leading to artificial lock contention. We term this a *synchronization attack*.

Second, if other threads also access the expanded structure, they will be forced to unnecessarily spend time in the critical section, thereby elongating the critical section. Worse, other threads competing to access the lock will have longer wait times. We term this a *framing attack* where a malicious thread manages to make other threads unnecessarily spend time accessing the expanded structure and also wait longer to acquire the lock.

In both synchronization and framing attacks, we observe the adversarial aspects of lock usage where a malicious thread can monopolize the lock usage and launch attacks simply by expanding the data structure tremendously.

In a cooperative environment, where all the threads belong to a single program, user, or organization, the lock usage concern is real. However, the effects of scheduler subversion and adversarial synchronization can be mitigated due to the cooperative nature of the entities involved. The developer can orchestrate the lock usage as they see fit, either by rewriting the program to reduce lock usage or using a different lock.

In a shared or competitive environment, such as servers, operating systems, or hypervisors, programs from multiple tenants may run on the same physical hardware. Such shared environments are commonly found in data centers and cloud computing settings. As anyone can be a tenant, including competitors or malicious actors, such installations place a

heavy burden on system software to isolate mutually distrusting tenants. The shared environment can either be benign where the actors are competitive but not malicious or hostile where the actors are competitive but can be malicious. In both these settings, different techniques need to be employed to enforce strong performance isolation.

In a benign setting, strong performance isolation can be guaranteed by the scheduling of resources. As the intent of the tenants is not malicious, one can safely assume that by careful allocation of resources, the desired isolation can be achieved. In the first part of this chapter, for benign settings, we show that the widespread presence of concurrent shared environments can greatly inhibit a scheduler and prevent it from reaching its goals. Infrastructure in such environments is commonly built with classic locks [39, 163, 165] and thus is prone to the scheduler subversion problem. As the data structure grows and shrinks during any workload execution, the critical section size keeps changing, leading to the scheduler subversion problem.

In the latter half of this chapter, for hostile settings, we show that the high degree of sharing across tenants through operating system data structures creates an avenue for performance interference via adversarial synchronization. We illustrate the performance interference on real-world applications in containers, as used by Docker [113]. Containers are highly efficient as they cost little more than an operating system process but rely on a shared operating system kernel between tenants with internally shared data structures. We show that even with an isolated environment provided by a container, malicious actors can access the shared data structures in the kernel tremendously.

Apart from the performance impact due to the attack, tenants may also suffer economically. Most of the cloud providers charge the users based on CPU usage. With the synchronization attacks, as the tenants wait to acquire the lock, they may spin if the lock is a spinlock. With longer wait

times, they may spin longer without doing any useful work and still be charged for the CPU usage. With the framing attacks, as the tenants are forced to access the expanded data structure, their CPU usage increases too, and they will be charged for the extra CPU usage.

The rest of this chapter is organized as follows. In Section 3.1, using a real-world application for a benign setting, we show the problem of scheduler subversion and discuss why scheduling subversion occurs for applications that access locks. In Section 3.2, for a hostile setting, we discuss the security aspects of synchronization and describe how exploiting the synchronization mechanisms can lead to denial-of-service attacks. Finally, we summarize in Section 3.3.

3.1 Scheduler Subversion

In this section, we discuss scheduler goals and expectations in a shared environment. We describe how in a benign setting, locks can lead to scheduler subversion in an existing application.

3.1.1 Imbalanced Scheduler Goals

Scheduling control is desired for complex multi-threaded applications and shared services. Operating system schedulers are responsible for a variety of goals, from utilizing hardware resources effectively to guaranteeing low latency for latency-sensitive applications [27, 59, 134, 159]. In this thesis, our focus is on CPU scheduling only.

In an ideal system, the OS scheduler and its specific policies determine the share of CPU time each running thread obtains. Classic schedulers, such as the multi-level feedback queue [21, 59], use numerous heuristics to achieve performance goals such as interactivity and a good batch throughput [134]. Other schedulers, ranging from lottery schedul-

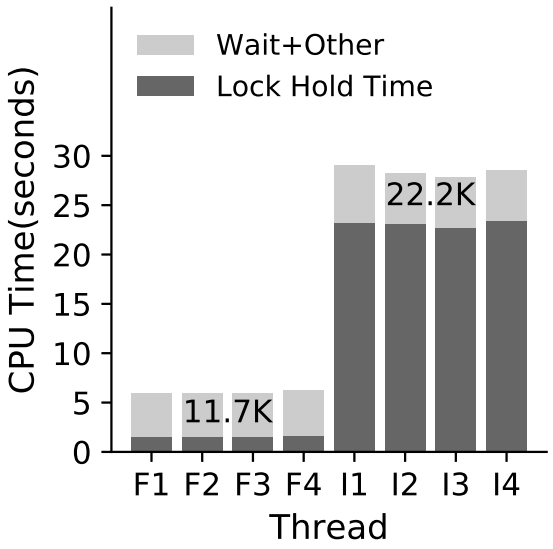


Figure 3.1: **Scheduler subversion with UpScaleDB.** *We use a modified benchmarking tool ups_bench available with the source code of UpScaleDB to run our experiments. Each thread executes either find or insert operations and runs for 120 seconds. All threads are pinned on four CPUs and have default thread priority. "F" denotes find threads while "I" denotes insert threads. "Hold" represents the critical section execution, i.e., when the lock is held; "Wait + Other" represents the wait-times and non-critical section execution. The value presented on the top of the dark bar is the throughput* (*operations*/*second*).

ing [159] to Linux CFS [27], aim for proportional sharing, thereby allowing some processes to use more CPU than others.

Unfortunately, current systems are not able to provide the desired control over scheduling. To illustrate this problem, we conduct an experiment using UpScaleDB [155] while running a simple analytical workload. We perform the experiment on a 2.4 GHz Intel Xeon E5-2630 v3 having two sockets; each socket has eight physical cores with hyper-threading enabled. The machine has 128 GB RAM and one 480 GB SAS SSD. The machine runs Ubuntu 16.04 with kernel version 4.19.80, using the CFS

scheduler.

We design an experiment where threads repetitively try to acquire the lock in a real-world application to confirm our hypothesis that lock usage imbalance can lead to scheduler subversion. Each thread executes a particular type of operation so that two different threads are executing variable-sized critical sections. If the usage of the locks leads to scheduler subversion, the threads spending more time in the critical section will be allocated more CPU than the other threads.

We use a modified built-in benchmarking tool ups_bench available with the source code of UpScaleDB to generate the workload. The workload comprises eight threads, of which four threads execute insert operations while the remaining four threads execute find operations. We pin all these eight threads on four CPUs and set the thread priority of each thread to the default. We run the workload for 120 seconds. We measure the throughput, i.e., the number of operations completed by each thread and the total CPU time allocated to each thread. Additionally, we also measure the time spent by each thread in the critical section while holding the global pthread mutex lock used by UpScaleDB to protect the environment state.

Given that all the eight threads have default thread priority, the expectation is that the CFS scheduler will allocate CPU equally to each thread. Figure 3.1 shows the CPU time allocated to each thread, the amount of time spent in the critical section by each thread, and its throughput. Contrary to our expectation, we observe that not all threads are allocated an equal amount of CPU. Although the desired CPU allocation for each thread is the same, the insert threads are allocated significantly (nearly five to six times) more CPU than the find threads leading to an imbalance in the set scheduling goals.

Another interesting observation to note in the figure is that the insert threads hold the global Pthread mutex lock significantly longer than the

find threads. The majority of CPU time is spent executing critical sections as we observe that the global lock is utilized for 80% of the experiment time, creating enough contention. When contention happens, the waiting threads block until the lock is free. However, as the workload continuously issues insert and find operations before the waiting thread wakes up to acquire the lock, another thread acquires the lock forcing the just woken thread to block again.

The global lock protects the B+tree data structure used by UpScaleDB. As the B+tree grows during the experiment, the critical section size varies. Therefore, the insert threads dominate the lock usage compared to the find threads, thereby dominating the CPU allocation too, leading to an imbalance in the CPU allocation. We call this imbalance the *scheduler subversion* problem where instead of the CPU scheduler determining the share of the CPU each thread obtains, the lock usage pattern dictates the share.

3.1.2 Non-Preemptive Locks

The scheduler subversion we saw in UpScaleDB is largely due to how UpScaleDB uses the global Pthread mutex lock. Locks are a key component in the design of multi-threaded applications, ensuring correctness when accessing shared data structures. With a traditional non-preemptive lock, the thread that holds the lock is guaranteed access to read or modify the shared data structure; any other thread that wants to acquire the lock must wait until the lock is released. When multiple threads are waiting, the order they are granted the lock varies depending on the lock type; for example, test-and-set locks do not guarantee any particular acquisition order, whereas ticket locks [109] ensure threads take turns acquiring the given lock.

For our work, we assume the presence of such non-preemptive locks to guard critical data structures. While great progress has been made

in wait-free data structures [75] and other high-performance alternatives [52], classic locks are still commonly used in concurrent systems, including the operating system itself.

3.1.3 Causes of Scheduler Subversion

Scheduler subversion problem arises in a concurrent system when:

- Condition SS1: Presence of a mutual exclusion lock that may block and is protecting a data structure.
- Condition SS2: Critical sections are significantly different lengths. We call this *different critical-section lengths*.
- Condition SS3: Time spend in the critical sections is high, and there is significant lock contention. We call this *majority locked run time*.

Condition SS1 is common as the majority of the concurrent systems still rely on mutual exclusion locks for synchronization [39, 70, 163, 165]. There are two reasons why scheduling subversion occurs for applications that access locks. First, scheduler subversion may occur when critical sections are of significantly different lengths: when different threads acquire a lock, they may hold the lock for varying amounts of time. We call this property *different critical-section lengths*. Secondly, scheduler subversion may occur when the time spend by the threads in the critical sections is high leading to significant lock contention. We call this property *majority locked run time*.

Condition SS2 of different critical-section lengths holds in many common use cases. For example, imagine two threads concurrently searching from a sorted linked list. If the workload of one thread is biased towards the front of the list, while the other thread reads from the entire list, the second thread will hold the lock longer, even though the operation is identical. Similarly, the cost of an insert into a concurrent data structure

Applications	Workload Notes
Memcached (Hashtable)	Get and Put operations are executed, having 10M entries in the cache using the memaslap tool.
Leveldb (LSM trees)	2 threads issuing Get, and 2 threads issuing Put operations are executed on an empty database using the db_bench tool.
UpScaleDB (B+ tree)	1 thread issuing Find and 1 thread issuing Insert operations are executed on an empty database.
MongoDB (Journal)	Write operations are executed on an empty collection. Write concern w = 0, j = 1 used. Size of operations is 1K, 10K and 100K respectively.
Linux kernel (Rename)	First rename is executed on an empty directory while the second rename is executed on a directory having 1M empty files; each filename is 36 characters long.
Linux kernel (Hashtable)	Design similar to the futex table in the Linux kernel that allows duplicate entries to be inserted and delete operation deletes all the duplicate entries in the hashtable.

Table 3.1: **Application & Workload details.** *The table shows the workload details for various user-space applications and the Linux kernel that we use to measure the critical section lengths.*

is often higher than the cost of a read; thus, one thread performing inserts will spend more time inside the critical section than another thread performing reads.

We measure how the length of the critical sections varies in real-world applications by running experiments involving a variety of user-space applications like Memcached, Leveldb, UpScaleDB, MongoDB, and Linux kernel that use different data structures.

We choose applications like Memcached, MongoDB as they can be used as servers serving key-value data; LevelDB and UpScaleDB can be used to build similar servers hosting key-value data; Linux kernel is used in Cloud environments to run multiple VM's and containers on a single host. All of these are designed using different data structures, as shown in Table 3.1 and support variety of operations having different sizes.

The Linux kernel rename system call operates on directories while

locking the entire filesystem to avoid deadlocks. A filesystem workload may comprise of rename operations where the directory sizes may vary. Similarly, the futex table in the Linux kernel is used for thread and process synchronization in user-space. Multiple user-space applications may use the same futex table leading to interference between the applications.

The workload involves executing different types of operations (get/find or put/insert) and that have different sizes of operations. Table 3.1 summarizes the workload details. As shown in Table 3.2 the lengths of critical sections do vary in real-world applications.

We note two important points. First, the same operation within an application can have significant performance variation depending on the size of the operation (e.g., the size of writes in MongoDB) or the amount of internal state (e.g., the size of a directory for renames inside the Linux kernel). Second, the critical section times vary for different types of operations within an application. For example, in leveldb, write operations have a significantly longer critical section than find or get operations.

For condition *SS3*, there are many instances in research literature detailing the high amount of time spent in critical sections; for example, there are highly contended locks in Key-Value stores like UpScaleDB (90%) [70], KyotoCabinet [55, 70], LevelDB [55], Memcached (45%) [70, 98]; in databases like MySQL [70], Berkeley-DB (55%) [98]; in Object stores like Ceph (53%) [117]; and in file systems [115] and the Linux kernel [30, 162]. The % in the bracket indicates the total time of the runtime spent in the critical section.

Unfortunately, different critical section lengths combined with spending a significant amount of time in critical sections directly subverts scheduling goals. When most time is spent holding a lock, useful CPU usage is determined by lock ownership rather than by scheduling policy: the algorithm within the lock to pick the next owner determines CPU usage instead of the scheduler. Similarly, when locks are held for different

Applications	Operation Type	LHT Distributions (microseconds)				
		Min	25%	50%	90%	99%
memcached	Get	0.02	0.05	0.11	0.16	0.29
(Hashtable)	Put	0.02	0.04	0.10	0.14	0.28
leveldb	Get	0.01	0.01	0.01	0.01	0.02
(LSM trees)	Write	0.01	0.11	0.44	4,132.7	43,899.9
UpScaleDB	Find	0.01	0.01	0.03	0.24	0.66
(B+ tree)	Insert	0.36	0.87	1.11	4.37	9.55
MongoDB (Journal)	Write-1K	230.3	266.7	296	497.2	627.5
	Write-10K	381.6	508.2	561.6	632.8	670.3
	Write-100K	674.2	849.3	867.8	902.4	938.9
Linux Kernel	Rename-empty	1	2	2	3	3
(Rename)	Rename-1M	10,126	10,154	10,370	10,403	10,462
Linux Kernel	Insert	0.03	0.04	0.04	0.05	0.13
(Hashtable)	Delete	0.06	1.15	1.61	9.27	18.07

Table 3.2: **Lock hold time (LHT) distribution.** *The table shows LHT distribution of various operations for various user-space applications and the Linux Kernel that use different data structures.*

amounts of time, the thread that dwells longest in a critical section becomes the dominant user of the CPU.

For the UpScaleDB experiment discussed earlier, we see that these two properties hold. First, insert operations hold the global lock for a longer period than the find operations, as shown in Table 3.2. Second, the insert and find threads spend around 80% of their time in critical sections. Thus, under these conditions, the goals of the Linux scheduler are subverted, and it cannot allocate each thread a fair share of the CPU.

3.2 Synchronization under Attack

This section shifts our focus to a hostile setting where malicious actors can deliberately expand the data structures to launch synchronization and framing attacks. Note that in the previous section, we discussed how when the data structure grows naturally, there is a likelihood that

the critical section size will vary. We will now see how a malicious actor can deliberately expand the data structure to vary the critical section size. Firstly, we discuss how shared infrastructure in data centers relies on shared data structures protected by varied synchronization mechanisms. Then, we show how a malicious actor can exploit the synchronization mechanisms to launch denial-of-service attacks.

Container-based isolation is quickly picking up pace as one of the most common tenants environment for shared infrastructure. With containers, an operating system kernel provides virtual software resources (files, sockets, processes) to each container. Substantial effort has gone into isolation so that the one container or VM cannot affect the performance of others [47, 72, 81, 100, 112, 143, 158] and each container or VM obtains a fair share of the resources. Our work focuses on container-based isolation, as the higher-level interfaces create more opportunities for performance interference.

An operating system should ideally execute containers in a perfectly isolated environment. In reality, though, container isolation is based on a combination of mechanisms. For preemptable resources, containers rely on the CPU, disk, or network scheduler to fairly share the resource between containers and prevent monopolization. For memory, accounting and allocation limits prevent containers from overusing memory. The operating system provides private namespaces for each container that prevents them from accessing resources of other containers, such as private file system directory trees and private sets of process IDs.

However, these isolation controls are built atop shared kernel data structures; in many cases, the kernel maintains global data structures shared by all containers and relies on scheduling, accounting, and namespaces to prevent any interference.

3.2.1 Synchronization and Framing Attacks

Container isolation mechanisms do not directly isolate access to the operating system's global data structures. Operating system kernels contain hundreds of data structures global to the kernel and shared across containers. These structures rely on synchronization primitives such as mutual exclusion locks, read copy update (RCU), and reader-writer(RW) locks to allow concurrent access. Multiple containers make unprivileged system calls in a shared environment to access kernel data structures using these synchronization primitives. We primarily focus on the mutual exclusion locks and RCU as they are heavily used in the kernel.

Synchronization primitives do not control how long one tenant can spend in the critical section accessing the data structure. Locks are mutually exclusive such that once held, they prevent any other process trying to acquire the lock from making progress. RCU allows multiple readers to access the data structure, but updaters wanting to free objects must wait until all prior read critical sections complete [108]. We call the time to wait to either acquire the mutual exclusion locks or to let all the prior read critical sections complete *synchronization stalls*.

Consider a linked list supporting insert and find operations protected by a lock as shown in Listing 3.1. An attacker can cause simple lock contention by repeatedly accessing the list. If the list is short, the synchronization stalls will not be that long. However, if an attacker can cause entries to be added to the list, the time spent in the find operations increases; on adding a million entries to the list, traversing the list will take a long time to complete, during which victims stall waiting to access the list.

We term this attack a *synchronization attack*, in which an attacker manages to increase the critical section size protected by synchronization primitives to deny victims access to one or more shared data structures. Such an attack occurs when:

- Condition S1: A shared kernel data structure is protected by a synchronization primitives such as mutual exclusion locks or RCU that may block.
- Condition S2: Unprivileged code can control the duration of the critical section by either
  - S2_input: providing inputs that cause more work to happen within the critical section
    OR
  - S2_weak: accessing a shared kernel data structure with weak complexity guarantees e.g., linear.
    AND
  - S2_expand: expanding or accessing the shared kernel data structure to trigger the worst-case performance.

We term the case when an attacker exploits condition S2_input by using large input parameters to increase the critical section size an *input parameter attack*. We will show in Chapter 4 how an attacker can execute a rename operation on a large directory, which holds a shared per-filesystem lock while traversing the entire directory. Additionally, during our investigation, we find that Apparmor [19] holds a shared namespace root lock while loading profiles, so loading a large profile can hold the lock for tens of seconds.

Algorithmic complexity attacks are a class of denial-of-service attacks that exploit the weak complexity guarantees of the data structures used to build many common applications. Data structures such as hash tables or unbalanced binary trees have better average-case performance than the worst-case performance. However, with certain inputs, these data structures can exhibit worst-case performance when subjected to certain inputs leading to poor performance and denial-of-service [46].

When the attacker launches an algorithmic complexity attack, they exploit the S2_*weak* and S2_*expand* conditions. For example, with both locks and RCU, for the list in Listing 3.1, an attacker can first add millions of entries and then switch to a workload that traverses the list. With RCU, victims deleting from the list stall, waiting for the attacker to complete its lengthy read-side critical section.

The synchronization attack is an *active attack* as the attacker needs to participate in executing a long critical section. However, in some cases, the attack can continue without the participation of the attacker. For example, consider what can happen if other tenants traverse the elongated list. After an attacker adds millions of entries to the list, other processes will continue to traverse the longer list, leading to both more time traversing the list and more time stalling on the lock.

We term this a *framing attack* because an inspection of who holds the lock will incorrectly frame innocent victim threads rather than identifying the attacker that expanded the data structure. This is similar to the framing in crime where the perpetrator is trying to frame someone else for their crime. This is a *passive attack*, as the attacker needs to do nothing to continue the performance degradation. More precisely, a framing attack is an extension and refinement on a synchronization attack and occurs when:

- Condition S1 + S2_*weak* + S2_*expand*: An attacker manages to expand a shared kernel data structure with weak complexity guarantees, i.e., a synchronization attack is in progress or was launched earlier.

- Condition F1: Victim tenants access the affected portion of the shared data structure with worst-case behavior.

In framing attacks, for mutual exclusion locks, the excessive stalls are attributed to other victims traversing the list rather than the attacker that

grew the list. RCU relies on the *grace period* to ensure that existing readers accessing the item being deleted have since dropped their references before a delete operation starts. For expanded data structures, the longer read-side critical section will lead to a longer grace period leading to poor performance. Thus, the victims continue to observe poor performance due to the past actions of the attacker and may be blamed for poor performance.

Framing and synchronization attacks can happen at the same time. Consider a situation where a hash table uses the protected list to build hash buckets. The attacker may target a single hash bucket by adding many entries leading to a synchronization attack. The victims will have to wait longer to acquire the lock. If one of these starved victims access the target hash bucket, they will traverse the elongated list and hold the lock longer, leading to a framing attack. Thus, synchronization attacks make the victims stall longer, while the framing attacks make the victims spend more time in the critical section.

Algorithmic Complexity Attacks vs. Adversarial Synchronization. Even though synchronization and framing attacks look similar to algorithmic complexity attacks, there is a fundamental difference between both these attacks.

There have been numerous algorithmic complexity attacks on Webservers [1, 144], XML parsers [8, 9], PHPMailer [4], Samba [2], Zens PHP engine [6], FTP [3], Red Hat directory server [7], DNS proxy servers [5] that will eventually end up exhausting one or more CPUs in the system. As the CPUs are exhausted; they cannot execute the regular user workload leading to denial-of-services.

However, for example, if a web server is running as a container where only a few CPUs are allocated to the container, a regular expression denial-of-service attack will only impact one container. As container isolation can guarantee proper isolation of resources like CPU, memory, disk,

and network, algorithmic complexity attacks may not impact all the containers running on the host. As all these resources are not shared amongst all the tenants, the impact of the attacks is restricted to the single tenant only.

On the other hand, synchronization and framing attacks target the synchronization primitives that are part of the shared kernel data structures. When a shared data structure is subject to either synchronization or framing attack, more than one container that needs to access the shared kernel data structures will observe longer stalls leading to poor performance or denial-of-services.

While the algorithmic complexity attacks target preemptable resources such as CPU, disk, or network, synchronization and framing attacks target non-preemptable resources like mutual exclusion locks. Existing container isolation mechanisms can effectively handle preemptable resources. However, the isolation mechanisms cannot handle the monopolization of the non-preemptive resources.

3.2.2 Threat Model

We now clearly define what the hostile environment is and present our assumptions about the malicious actor. One or more containers run on a single physical machine. All containers, including the one that plays the role of an adversary, hereafter called an attacker, run arbitrary workloads that can access OS services via system calls. An attacker may have thousands of users within a container. However, for the sake of simplicity, we assume there is a 1-1 mapping between tenants to users and each container is associated with a user. No container, including the attacker, has special privileges. Due to random cloud scheduling, we assume a single attacker is present, thereby removing the possibility of collusion. We place no limit on the number of containers a single user can run on a single physical machine.

The attacker's goal is to target synchronization primitives within an operating system such that other containers accessing the same primitives starve, leading to poor performance or denial-of-service. The attacker can use either a single container or multiple containers to launch one or more attacks.

We now examine synchronization and framing attacks on the Linux kernel when using containers for isolation and demonstrate three different attacks on three kernel data structures accessed by common system calls.

3.2.3 Synchronization and Framing Attacks on Linux Kernel

We describe here three Linux kernel data structures that are vulnerable to Algorithmic Complexity Attacks (ACAs) and can be used to launch synchronization and framing attacks. While ACAs are not new, we show how synchronization amplifies their effect. The summary of the attacks is shown in Table 3.3.

In the table, one can see the diversity of the attacks. Remember that the attacker is launching the attack without executing any privileged code. We are targeting different types of data structures designed for different purposes and use synchronization mechanisms differently. We also show three different ways to launch synchronization and framing attacks. There is no fixed way that the attacker has to use to launch attacks. We just show three attacks in this thesis. The Linux kernel comprises hundreds of different data structures, and hence there is a possibility that other data structures can also be targeted to launch the attacks.

We perform our experiments on a 2.4 GHz Intel Xeon E5-2630 v3 having two sockets; each socket has eight physical cores with hyper-threading enabled. The machine has 128 GB RAM and one 480 GB SAS SSD. The machine runs Ubuntu 20.04 having kernel version 5.4.62. All the

Data structure	Attack Type	Synchronization Primitive	Attack Strategy
Inode cache	Synchronization attack	Global spinlock	Break the hash function then run dictionary attack by creating thousands of files in the targeted hash bucket.
Futex table	Framing attack	Array of spinlocks	Probe hash buckets to identify one or more target hash buckets. Park thousands of threads on the hash bucket.
Dcache	Synchronization attack	RCU	Either break the hash function or overwhelm the hash buckets by randomly creating dcache entries.

Table 3.3: **Summary of the attacks.** *Brief description of the three attacks on the Linux kernel. While the inode cache and directory cache attacks are synchronization attacks, the futex table attack is a framing attack. Note the different methods that we use to launch the attacks.*

applications and benchmarking tools used for the experiments are run as separate Docker containers.

3.2.3.1 Synchronization Attack on Inode Cache

We now describe how an inode cache attack can be launched by first breaking the inode hash function. Then the attacker can target one or more hash buckets by creating thousands of files that map to the target hash bucket. The inode cache attack is a synchronization attack where the attacker actively participates in the lock acquisition process.

Inodes are filesystem objects such as regular files, directories, etc. They are stored in a filesystem, and on access are loaded in memory and live there. The Virtual File System maintains the inode cache to speed up file access by avoiding expensive disk accesses to read file metadata [13]. The inode cache is implemented as a hash table, and collisions are handled by storing a linked list of inodes with the same hash value in a hash bucket. A global lock `inode_hash_lock` protects the inode cache. The number of buckets in the hash table is decided at boot time based on memory size.[1]

The inode cache hash function combines the inode number, unique to each file, and the address of the filesystem superblock data structure in memory. This address is set when a volume is mounted but varies across systems and boots. While the inode number for a file is visible to unprivileged users, the superblock address is not, and without that address, it is hard to predict which hash bucket an inode will reside in.

We have found a way to break this function, which we describe in detail elsewhere [121]. By creating files with specific inode numbers, we determined that a user can probe for the superblock address, allowing them to create files in a single hash bucket that grows long and slow to

[1]For a 128 GB memory system used for evaluation, the inode cache has $2^{22} = 4,194,304$ hash buckets.

traverse. Generally, users cannot specify the inode number for a file as the file system chooses it. However, using a FUSE unprivileged file system in user-space [156], we design a file system implementation with complete control over inode numbers.

For Docker, mounting needs CAP_SYS_ADMIN which is privileged [77]. Linux supports unprivileged FUSE mounts[94], although Docker disables this by default.[2] As a workaround, we use the idea of Linux user namespaces [114] discussed by NetFlix to mount the FUSE file system in an unprivileged environment [60]. A user also suggests a similar workaround on the bug page that is filed to allow FUSE functionality by default. [2]

Linux namespaces allow per-namespace mappings of the user and group IDs. It enables a process's user to have two different IDs inside the user namespace and outside the namespace. Thus, a process will not be allowed privileged operations outside the namespace but has root privileges inside the namespace. We let the user within the container run with root privileges within its namespace; however, the user within the container cannot execute privileged operations outside the namespace from the host's perspective. This way, we mount FUSE filesystems without compromising the host's security measures.

After mounting the FUSE filesystem, a user can create files with arbitrary inode numbers and create collisions in the inode hash, leading to long lists in a hash bucket. Because of the large number of hash buckets, it is difficult for the attacker to target a specific file for contention. Instead, the attacker continues to access the same bucket, elongating critical sections while holding the global lock. As the lock is held for a longer duration, any other user trying to acquire the global lock must wait longer, leading to poor performance.

[2]A bug is already filed to allow FUSE functionality by default - https://github.com/docker/for-linux/issues/321.

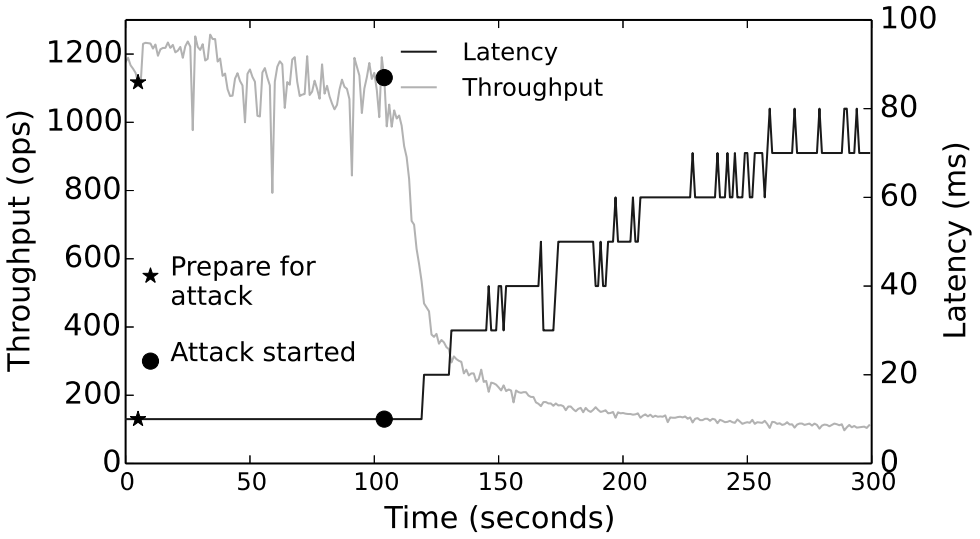


Figure 3.2: **Performance of Exim Mail Server under inode cache attack.** *Throughput and Average Latency timeline of Exim Mail Server when under inode cache attack. Prepare to attack means that the attacker starts to launch the attack and initiates probing to break the hash function and identify the superblock address. Upon identifying the superblock address, the attacker can target a hash bucket and launch an attack.*

In this attack, `inode_cache_lock` meets S1, hash table having linear worst-case complexity meets S2_*weak*, and FUSE filesystem meets S2_*expand* condition. Given enough resources and time, an attacker can turn this attack into a framing attack by identifying and targeting a hash bucket that victims often access.

To show the impact on the victim's performance, we run an Exim mail server container as a victim and launch an inode cache attack from a separate container. We run MOSBENCH [29] scripts as the client from another machine to send messages to the Exim server. To monitor the internal state of the inode cache, we track the lock hold times, wait times to acquire the lock for the victim and the attacker at the start of each second. Additionally, we also track the maximum number of entries for the victim

and the attacker in the buckets every 10 seconds.

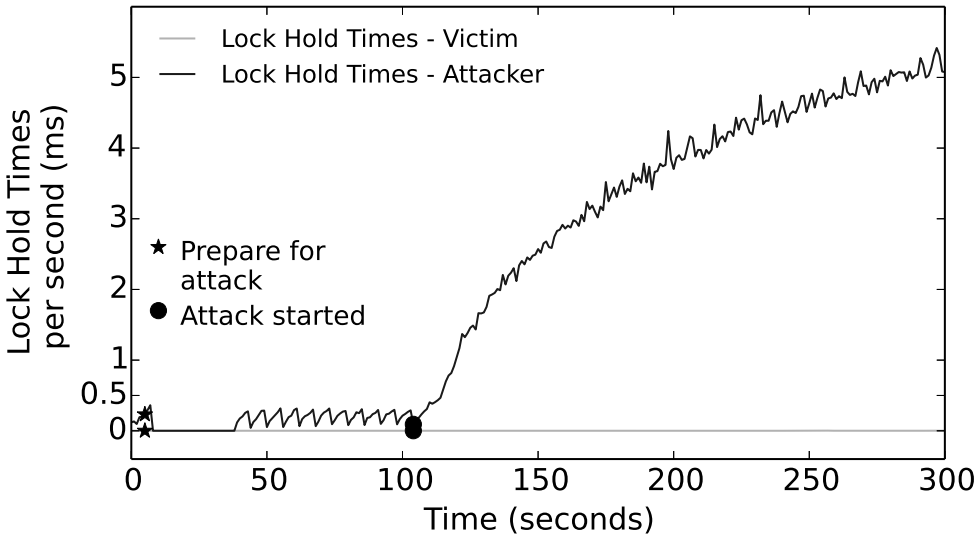
Figure 3.2 shows the timeline of the throughput and average latency for the duration of the attack. The attacker tries to identify the superblock address during the prepare phase by carefully choosing the inode numbers. After identifying the superblock address, the attacker can target any hash bucket and can start the attack. In the figure, around time 100, the attacker finds the superblock address.

Once the attack starts, the performance reduces significantly as the lock is held while adding entries to the targeted bucket, thereby starving the Exim mail server, and reducing the throughput by 92% (around 12X) and increasing the latency of the operations. As the attack progresses, the attacker continues to add more entries to the hash bucket, increasing the lock hold time further.

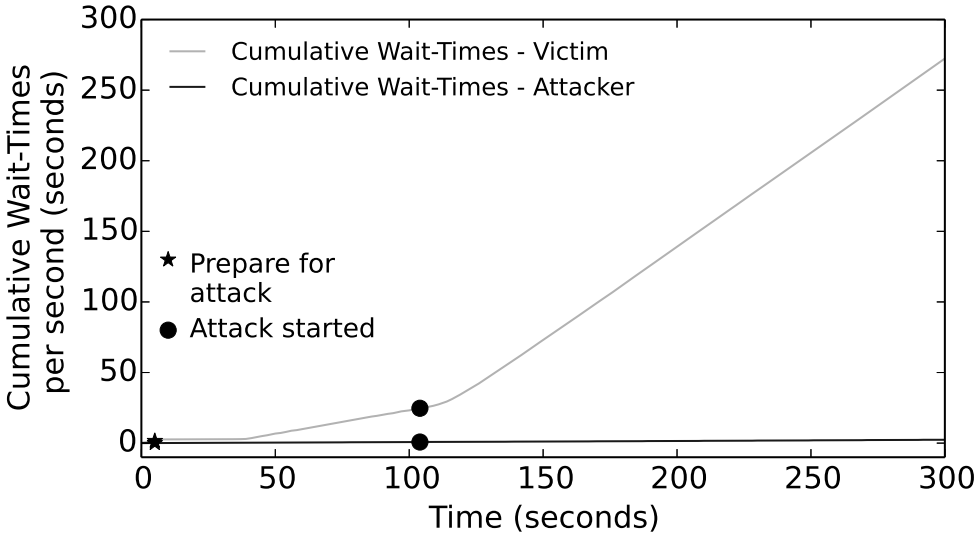
Internal state of the inode cache. Figure 3.3 shows the timeline of the lock hold times, the wait times, and the maximum number of entries for the victim and the attacker. In Figure 3.3a, we observe that once the attack starts, the attacker can elongate the lock hold times by targeting a single hash bucket and expanding it. As the victim is not accessing a single hash bucket, the lock hold times always stay around a few hundred nanoseconds.

Figure 3.3c corroborates the increase in the lock hold times as the number of entries keeps increasing as the attack progresses. As the victim's entries are spread across different hash buckets, the maximum number of entries in any bucket remains around two entries during the experiment.

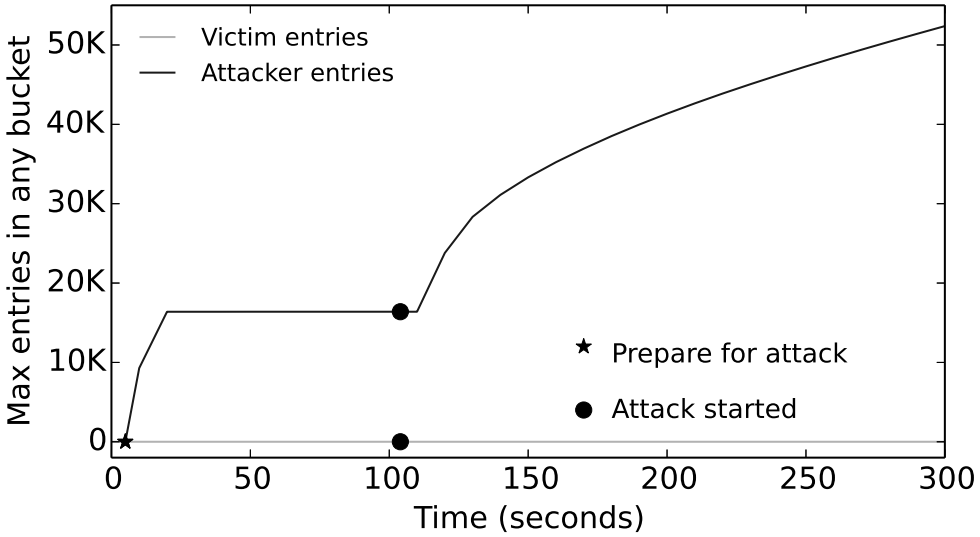
Figure 3.3b shows that the victim's cumulative wait times continue to grow as the attack progresses. Once the attacker starts dominating the lock hold times, the victim threads must wait longer to acquire the lock leading to poor performance. In Figure 3.3b, we observe that while dominating the lock hold times, the attacker does not have a tremendous increase in the cumulative wait times. This shows that the attacker is not



(a) Lock hold times of the victim and the attacker.



(b) Cumulative wait times of the victim and the attacker.



(c) Max number of entries of the victims and the attacker in any bucket.

Figure 3.3: **Internal state of inode cache when under inode cache attack.** *The graph present an overall picture of the inode cache when an attacker is launching the attack. In particular, the timeline shows the lock hold times, the cumulative wait times to acquire the inode cache lock, and the maximum number of entries of the victim and the attacker in the inode cache. The victim is running the Exim Mail Server.*

impacted by the victim's small lock hold times. It is only the victim that is impacted by longer wait times.

Economic impact. For the inode cache attack, once the attack starts, we observe that the victim threads' cumulative wait-time is around 273 seconds which is roughly around 33% of the total CPU used by the victim threads. Hence, the victim will have to pay 33% more for the CPU usage even though they are not doing any useful work. We believe that with a multi-threaded attacker, the performance and economic impact will be higher as multiple attacker threads will further increase the wait times.

3.2.3.2 Framing Attack on Futex Table

We now describe a framing attack on the futex table where the attacker probes the futex table hash buckets to find a target hash bucket. Upon finding a target hash bucket used by victims, the attacker will create thousands of threads and park them on the hash bucket by calling futex syscall. After targeting the hash bucket, the attacker no longer participates in the lock acquisition process in this attack.

The Linux kernel supports futexes, a lightweight method to support thread synchronization in user-space [63]. A futex provides the ability to wait on a *futex variable*, which is any location in memory until another thread signals the thread to wake up. Futexes are used to build synchronization abstractions such as POSIX mutexes and condition variables. The `futex()` syscall lets the user-space code wait and signal futex variables.

Rather than maintain a separate wait queue for each futex variable, the kernel maintains a *futex table*, and each bucket in the table is a shared wait queue. To identify the wait queue for a futex variable, the kernel hashes the futex variable address. When a thread waits on a futex, the kernel adds the thread to the wait queue dictated by the hash of the futex variable. Similarly, the kernel traverses the shared wait queue when waking a thread, looking for threads waiting on that futex variable. As a result, a single wait queue can be shared by several futex variables, either belonging to the same or different applications. A separate lock protects each hash bucket. Linux kernel allocates the futex table at boot time, and the number of buckets is a multiple of the number of CPUs in the system.[3]

As the number of hash buckets is small, the attacker uses bucket probing to identify a target hash bucket instead of breaking the hash function. The attack starts by allocating a few thousand futex variables to map them to different wait queues. The attacker then probes the wait queues by call-

[3]For the 32 CPU system used for evaluation, the hash table comprises $256 * 32 = 8,192$ hash buckets.

ing futex() to wake a thread for each while measuring the time it takes to complete the system call. The syscall will take measurably longer to complete if victim processes are already using a wait queue, allowing the attacker to attack these wait queues.

After identifying the wait queue, the attacker spawns thousands of threads that wait on the target futex variable, thereby adding them to its wait queue. As the wait queue grows, any victims sharing it must walk the elongated queue to wake up their threads. This leads to longer lock hold times, longer stalls to acquire the lock, and poor performance.

Unlike the inode cache attack, in this scenario, the attacker becomes passive and sits idle after creating the waiting threads, which demonstrates a framing attack – there is lock contention, but the attacker is not actively acquiring the lock. In this attack, the attacker meets the condition $S1 + S2\_weak + S2\_expand$ by parking thousands of threads on the target hash bucket. When the victim accesses the target hash bucket, the condition F1 is met.

We conduct an experiment by running UpscaleDB, an embedded key-value database [155], within a container as a victim to show the performance impact. We use the built-in benchmarking tool ups_bench to run an in-memory insert-only workload. Figure 3.4 shows the throughput and average latency timeline. Before the attack starts, UpscaleDB observes high throughput while the average latency remains constant.

During the first part of the attack, the attacker probes the futex table, and around time 54 seconds, identifies a busy-wait queue and starts creating threads to lengthen the queue. Note that the wait queue found is the one used by UpScaleDB to park the threads that are waiting to acquire the global lock that UpScaleDB uses for synchronization. This leads to highly variable performance for UpScaleDB, reducing throughput between 65 to 80%. Along with a drop in the throughput, we observe that the maximum latency increases from around 10-15 milliseconds to 0.7-1.2 seconds – an

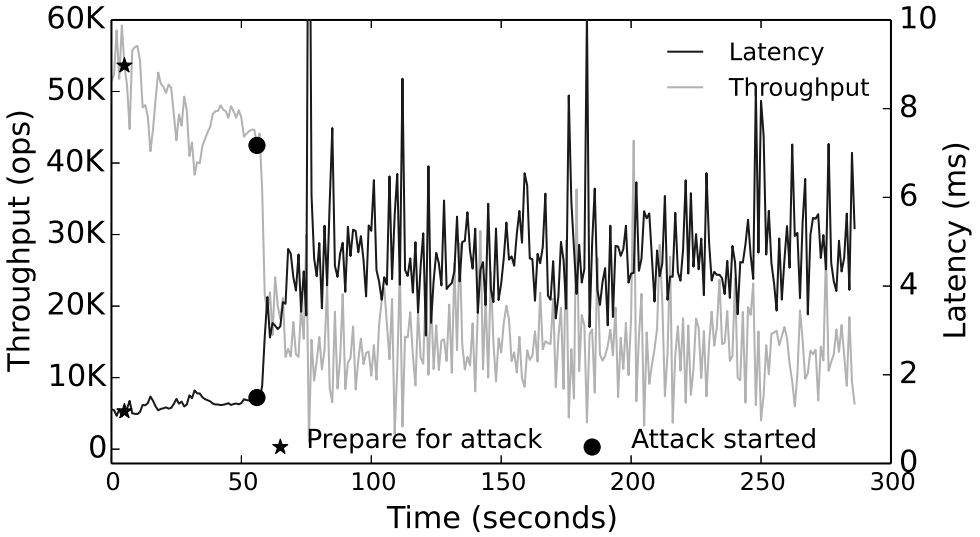
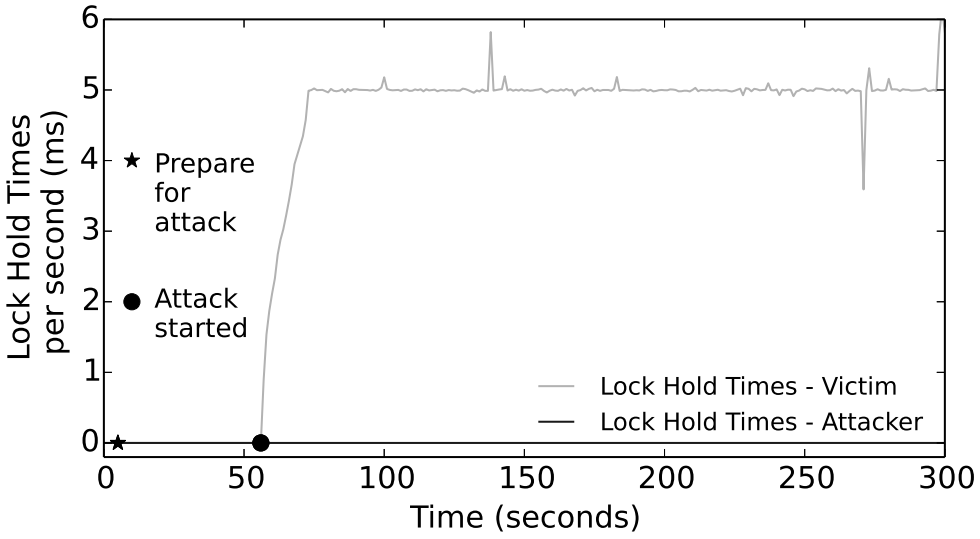


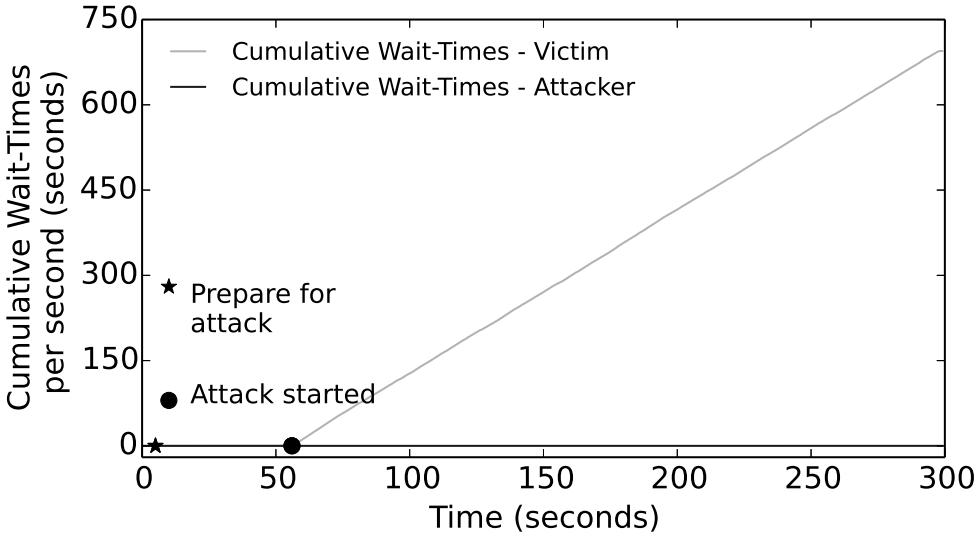
Figure 3.4: **Performance of UpScaleDB under futex table attack.** *Throughput and Average Latency timeline of UpScaleDB when under futex table attack. Prepare to attack means that the attacker starts to launch the attack and initiates probing to identify the hash bucket that UpSCaleDB uses. After identifying the hash bucket, the attack is launched by spawning thousands of threads and parking them in the identified hash bucket.*

increase of 45 to 100X.

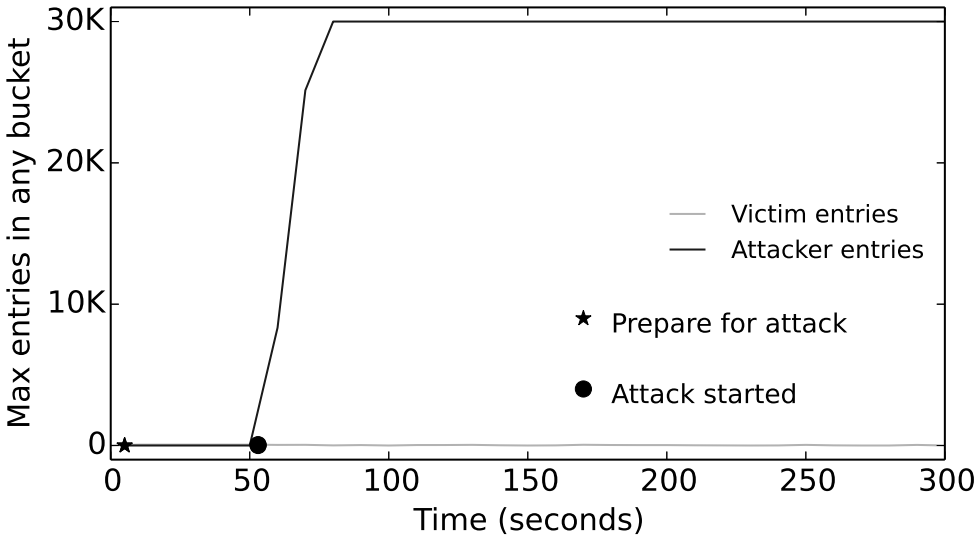
Internal state of the futex table. Figure 3.5 shows the timeline of the lock hold times, the wait times, and the maximum number of entries for the victim and the attacker. As the futex table attack is a framing attack, the attacker turns passive after launching the attack. In Figure 3.5a, we observe that once the attack starts, after elongating the hash bucket and parking thousands of threads in the wait queue, the lock hold times of the victim increases. As the victim thread must perform more work due to the attacker's parked threads, the lock hold times increase from a few hundred nanoseconds to tens of milliseconds. Being a framing attack, as the attacker is not participating in the lock acquisition process, the attacker's lock hold time is 0.



(a) Lock hold times of the victim and the attacker.



(b) Cumulative wait times of the victim and the attacker.



(c) Max number of entries of the victims and the attacker in any bucket.

Figure 3.5: **Internal state of futex table when under futex table attack.** *The graphs present an overall picture of the futex table when an attacker is launching the attack. In particular, the timeline shows the lock hold times, the cumulative wait times to acquire the hash bucket lock, and the maximum number of entries of the victim and the attacker in the futex table. The victim is running UpScaleDB.*

Figure 3.5c corroborates the increase in the victim's lock hold times once the number of entries increases after the attack is launched. Even though the victim's total entries is tiny, the victim still pays the cost of the elongated wait queue leading to a performance collapse.

Figure 3.5b shows that the victim's cumulative wait times continue to grow as the attack progresses. Being a framing attack, the attacker manages to increase the lock hold times of the victims and increase the cumulative wait times. As multiple victim threads participate in the lock acquisition process, each thread must wait longer to acquire the lock leading to an increase in the cumulative wait times.

Economic impact. Once the futex table attack starts, we observe that

the total cumulative wait-time for the victim is 694 seconds. The total cumulative wait time is roughly around 40% of the total CPU used by all the victim threads. Hence, the victim will have to pay 40% more for CPU usage due to the framing attack. The performance and the economic impact will increase when many victim threads participate in the lock acquisition process.

Framing attacks make the victims traverse the expanded hash bucket and unnecessarily expand the critical section increasing the victim's CPU usage. We observe an increase in the victim's total CPU usage by 2.3X times compared to the case without an attack. Thus, the victim may end up paying 2.3X times more for the extra CPU usage.

Even though UpScaleDB uses POSIX mutexes that are highly optimized to avoid the futex system call as much as possible, there is still a significant performance degradation. Note that by repeatedly parking and waking threads, the above attack will turn into a synchronization attack.

3.2.3.3 Synchronization Attack on Directory Cache

We now describe how a directory cache attack can be launched by breaking the hash function or simply creating millions of random dcache entries overwhelming the hash table. This attack targets RCU instead of the mutual exclusion locks and is a synchronization attack.

Lastly, we show a vulnerability that can be exploited by an attacker that can break the dcache hash function. One of the most common filesystem operations is a lookup, which maps a path string onto an inode residing in a memory. A dentry structure maps a path to an in-memory inode. The Linux directory cache (dcache) stores dentry structures to support filename lookups [153]. The dcache is implemented as a hash table where each bucket stores a linked list of dentries with the same hash value. The hash function uses the parent dentry address and the filename

to calculate the hash value.

For efficiency, the dcache relies on RCU to allow concurrent read access, but freeing entries must wait for all concurrent readers to leave the read critical section. This wait, called a *grace period*, ensures that no reader is holding a reference to the deleted object. RCU provides synchronous (`synchronize_rcu()`) or asynchronous (`call_rcu()`) APIs for this purpose. The synchronous API makes the user wait until the grace period is over. On the other hand, the asynchronous API registers a call back that the RCU subsystem executes after the grace period is over.

The dcache stores *negative entries* to record that no such on-disk file exists for a particular filename. One reason to allow negative entries is to support applications that issue lookups for a file on multiple paths specified by environment variables. Negative entries help avoid an expensive filesystem call for files that do not exist.

The attack exploits the dcache's support for negative entries. An attacker can create millions of negative entries mapping to a single hash bucket by breaking the hash function thereby meeting condition S1 + S2_*weak* + S2_*expand*. There has been an instance in the past where a customer observed performance degradation due to too many negative entries created by genuine workloads [67].

Before creating a negative entry, the lookup operation first walks through the hash bucket to check if the entry exists or not. The hash bucket walk is part of the RCU read-side critical section. Walking an expanded hash bucket increases the read-side critical section, thereby increasing the grace period size too. As RCU is shared across the Linux kernel, any increase in the grace period stalls the victims. Victims using the `synchronize_rcu()` will stall until the grace period is over. In the case of `call_rcu()`, freeing objects will be delayed, and more work will pile up for the RCU background thread to execute the callbacks leading to lower performance or out of memory conditions [104, 127, 128, 136].

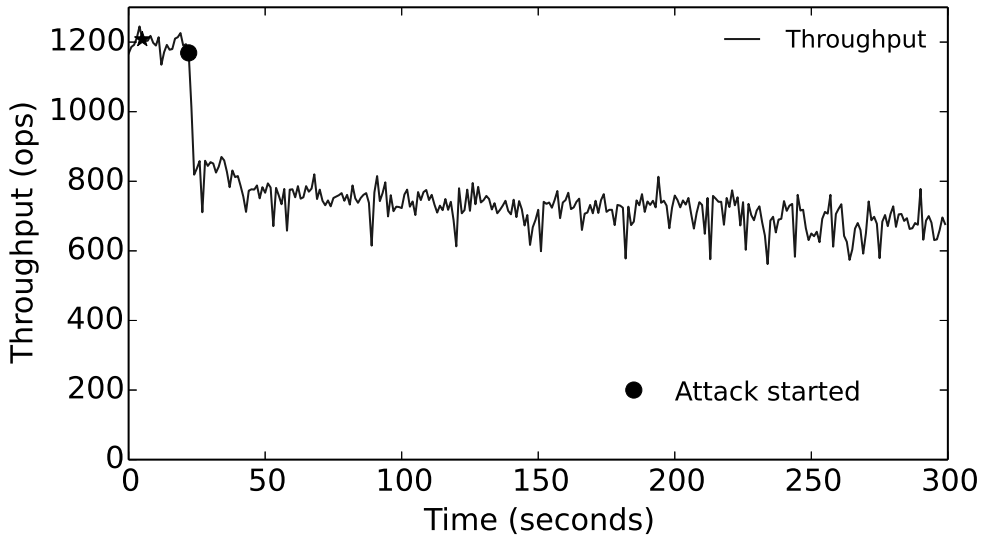


Figure 3.6: **Performance of Exim Mail Server under directory cache attack.** *Throughput timeline of Exim Mail Server when under directory cache attack. There is no need to prepare for the attack with directory cache attack. The attack starts immediately by creating millions of negative dentries.*

To demonstrate the attack and the impact on the victim's performance, we run an Exim mail server container as a victim and launch the directory cache attack from a separate container. We run the experiment by modifying the kernel to simulate an attacker that can target any hash bucket. In the past, the dcache hash function has been compromised [33] using birthday attack [68]. We can employ a similar methodology that we used to break the inode cache hash function to break the dcache hash function.

Figure 3.6 shows the throughput for the duration of the attack. Once the attack starts, as the hash bucket size increases, the read-critical section size increases, increasing the grace period size. We observe that the grace period increases from 30-40 milliseconds (no attack case) to around 1.7 seconds after the attacker runs for a few hours. The mail server workload generates hundreds of thousands of callbacks every second, which over-

whelms the RCU background thread when the grace period increases.

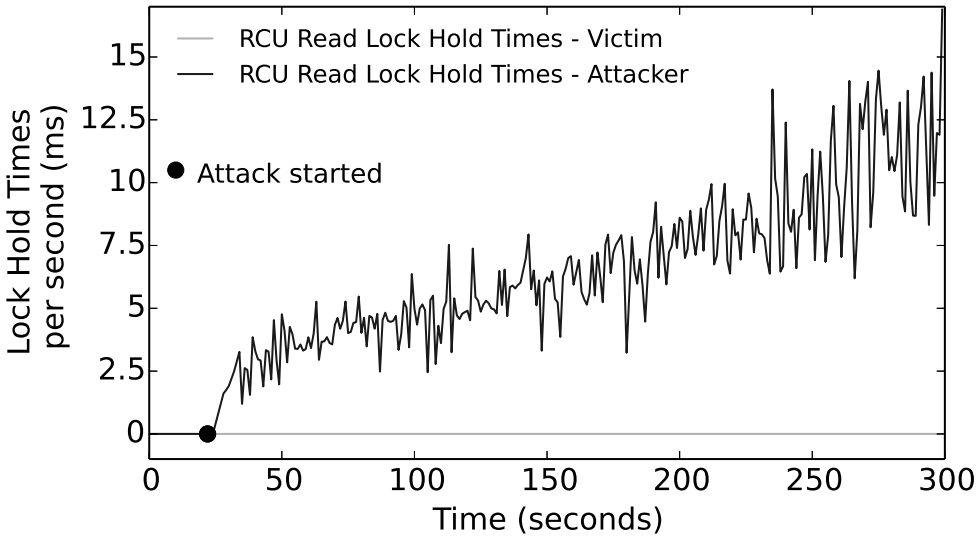
Internal state of the dentry cache. Figure 3.7 shows the timeline of the RCU read-side critical section times, the grace period duration, and the maximum number of entries for the victim and the attacker. In Figure 3.7a, we observe the read-side critical section sizes increase for the attacker's operations over time as each thread spends more time traversing the elongated hash bucket. On the other hand, the hash buckets accessed by the victims are not too long. Therefore, the read-side critical section size is tiny (a few hundred nanoseconds).

Figure 3.7c corroborates the increase in the attacker's critical section times once the number of entries increases after the attack is launched. On the other hand, the maximum number of entries for the victim is just two as all its entries are spread across the hash table.

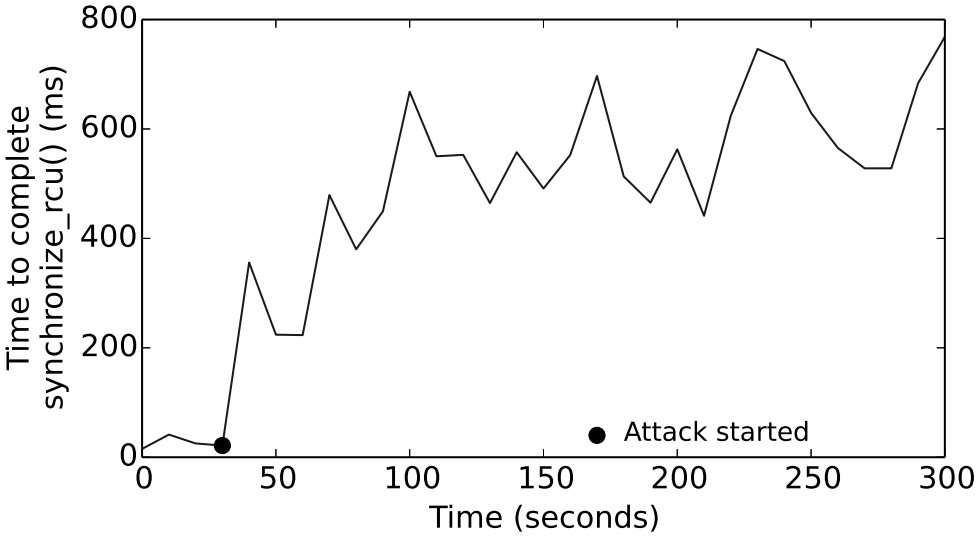
Lastly, Figure 3.7b shows the impact of the longer read-side critical sections on the duration of the grace period. With read-side critical section sizes increasing, the grace period also increases and reaches up to 790 milliseconds.

For the current experiment, the Exim mail server workload does not rely on synchronize_rcu(). However, it still initiates the call_rcu() calls to wait for the grace period to be over. Hence, it is not possible to gauge the economic impact of this workload. There may be other applications that might call synchronize_rcu() and hence will have to wait for the grace period to be over to make forward progress. There will be an economic impact in those situations like we have seen for the earlier two attacks. Moreover, as RCU is being shared across hundreds of subsystems in the Linux kernel; longer grace periods will impact various workloads.

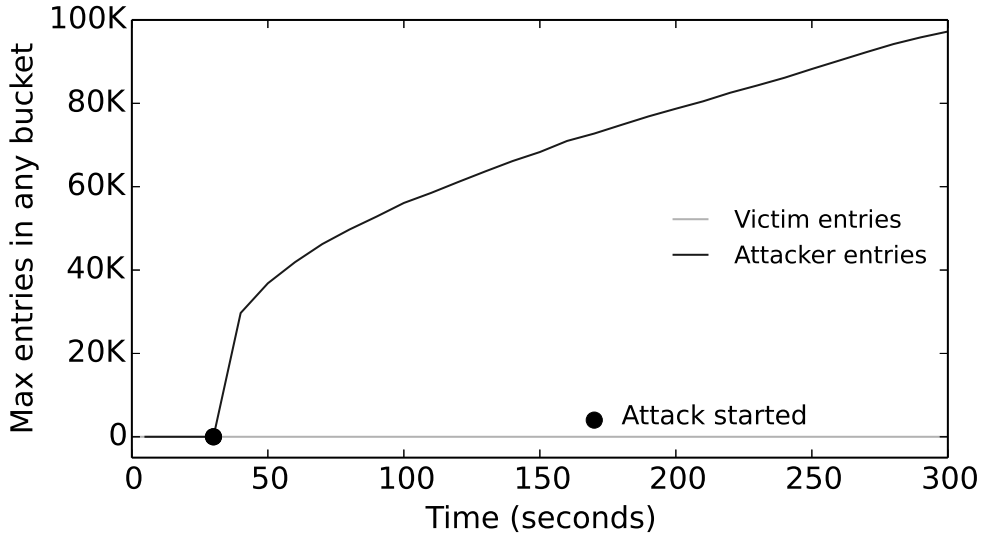
Alternate attack approach. An attacker can launch the same attack without breaking the hash function. Instead of targeting a single hash bucket, the attacker can randomly create hundreds of millions of negative entries stressing the hash table. We observe that by doing so, the



(a) Read-side lock hold times of the victim and the attacker.



(b) Time to complete the synchronize_rcu() call, i.e., the grace period time.



(c) Max number of entries of the victims and the attacker in any bucket.

Figure 3.7: **Internal state of dentry cache when under dentry cache attack.** *The graphs present an overall picture of the dentry cache when an attacker is launching the attack. In particular, the timeline shows the lock hold times for the read-side critical section, the time taken to complete synchronize_rcu() call, and the maximum number of the entries of the victim and attacker in the dentry cache. The victim is running the Exim Mail Server.*

grace period size increases to around 400 milliseconds within seconds of starting the attack leading to a drop in the victim's performance.

Through these three different attacks on three data structures, we show how attackers can leverage synchronization primitives to increase synchronization stalls, hurting the victim's performance. We also show that an attacker can use different methods to launch an attack – by breaking the hash bucket, by probing the hash tables when the number of hash buckets is not large, and by stressing the hash tables by creating objects randomly.

3.3 Summary & Conclusion

Locks are an important component of concurrent systems. In this chapter, we introduced a new important property of locks – lock usage. Lock usage deals with the amount of time spent in the critical section while holding the lock. As multiple threads participate in the lock acquisition process in a concurrent system, lock usage becomes crucial. Unfair lock usage can lead to two problems – performance and security.

We introduced two new problems that deal with locks and lock usage. Firstly, we introduced the problem of scheduler subversion where instead of the CPU scheduler determining which thread runs, the lock usage patterns dictate the CPU share, thereby subverting the scheduling goals. Due to scheduler subversion, the entire system may exhibit fairness and starvation problems leading to poor performance.

Secondly, through adversarial synchronization, we showed the adversarial aspects of lock usage and described two types of attacks – synchronization and framing attacks. By carefully accessing the data structures that the synchronization primitives protect, an adversary can control the lock usage leading to poor performance and large denial-of-services. By launching synchronization attacks, an adversary can make the victims stall longer to acquire the lock. On the other hand, a framing attack is an extension of a synchronization attack, where an adversary forces the victim to spend more time in the critical section while making other victims stall longer to acquire the lock.

When locks are used inside of a traditional multi-threaded program, the lock usage concern is real. But the effects of scheduler subversion and adversarial synchronization can always be mitigated due to the *cooperative* nature of the entities involved, i.e., the threads are all part of the same program. Thus, the burden of using locks correctly can be placed on the application developer; any scheduling subversion or adversarial synchronization hurts only the threads within that application.

However, when different clients in a competitive access locks environment (e.g., a server, the kernel, etc.), important locks maybe unintentionally or even maliciously held for significantly different amounts of time by different clients; thus, the entire system may exhibit fairness and starvation problems or denial-of-service attacks. In these cases, lock design must include mechanisms to handle the competitive nature of their usage.

4
Scheduler-Cooperative Locks

In the previous chapter, we discussed how using locks can lead to a new problem called scheduler subversion. When subversion arises, instead of the CPU scheduler determining the proportion of the processor each competing entity obtains, the lock usage pattern dictates the share. Irrespective of the CPU scheduling goals, locks drive the CPU allocation leading to performance issues. Today's datacenters are designed around concurrent shared infrastructure such as operating systems, hypervisors, or servers. Such infrastructure is commonly built with classic locks; hence can greatly inhibit a scheduler and prevent it from reaching its goal.

In this chapter, we now discuss how to address the problem of scheduler subversion by building locks that align with the scheduling goals. To remedy this problem, we define the concept of *usage fairness*. Usage fairness guarantees that each competing entity receives a time window in which it can use the lock (once or many times); we call this *lock opportunity*. By preventing other threads from entering the lock during that time, lock opportunity ensures that no one thread can dominate the CPU time.

To study usage fairness, we first study how existing locks can lead to the scheduler subversion problem. We then propose how lock usage fairness can be guaranteed by *Scheduler-Cooperative Locks* (*SCLs*), a new approach to lock construction. SCLs utilize three important techniques to achieve their goals: tracking lock usage, penalizing threads that use locks excessively, and guaranteeing exclusive lock access with lock slices.

By carefully choosing the size of the lock slice, either high throughput or low latency can be achieved depending on scheduling goals.

Our work focuses on locks that cooperate with proportional-share schedulers. We implement three different types of SCLs. The user-space Scheduler-Cooperative Lock (u-SCL) is a replacement for a standard mutex and the Reader-Writer Scheduler-Cooperative Lock (RW-SCL) implements a reader-writer lock; these are both user-space locks and can be readily deployed in concurrent applications or servers where scheduler subversion exists. The kernel Scheduler-Cooperative Lock (k-SCL) is designed for usage within an OS kernel. Our implementations include novel mechanisms to lower CPU utilization under load, including a *next-thread prefetch mechanism* that ensures fast transition from the current-lock holder to the next waiting thread while keeping most waiting threads sleeping.

Using microbenchmarks, we show that in a variety of synthetic lock usage scenarios, SCLs achieve the desired behavior of allocating CPU resources proportionally. We also study the overheads of SCLs, showing that they are low. We investigate SCLs at a small scale (a few CPUs) and a larger scale (32 CPUs), showing that behavior actually improves under higher load as compared to other traditional locks.

We also demonstrate the real-world utility of SCLs in three distinct use cases. Experiments with UpScaleDB [155] show that SCLs can significantly improve the performance of find and insert operations while guaranteeing usage fairness. Similarly, experiments with KyotoCabinet [93] show that, unlike existing reader-writer locks, RW-SCL provides readers and writers a proportional lock allocation, thereby avoiding reader or writer starvation. Lastly, we show how k-SCL can be used in the Linux kernel by focusing upon the global filesystem rename lock, *s_vfs_rename_mutex*. This lock, under certain workloads, can be held for hundreds of seconds by a single process leading to an input parameter

attack we described in the previous chapter. Such an attack starves other file-system-intensive processes. We show that k-SCL can be used to mitigate the lock usage imbalance and the input parameter attack.

The rest of this chapter is organized as follows. We first show in Section 4.1 that existing locks do not ensure lock usage fairness. Then we discuss the design and implementation of SCLs in Section 4.2 and evaluate the different implementations of SCLs in Section 4.3. We present limitations and applicability of SCLs in Section 4.4 and summarize in Section 4.5.

4.1 Lock Opportunity

In this section, we describe how existing locks do not guarantee lock usage fairness and introduce *lock opportunity* – a new metric to measure lock usage fairness.

4.1.1 Inability to Control CPU Allocation

Existing locks, such as mutex, spinlocks, and even ticket locks, do not enable schedulers to control the CPU allocation given to different threads or processes. We illustrate this with a simple application that has two threads and a single critical section. For one of the threads (T0), the critical section is long (10 seconds), while for the other (T1), it is short (1 second); for both, the non-critical section time is negligible (0). We consider three common lock implementations: a pthread mutex, a simple spinlock that uses the test-and-set instruction and busy-waits, and a ticket lock that uses the *fetch-and-add* instruction and busy-waits. The application runs for 20 seconds.

As shown in Figure 4.1, even though the scheduler is configured to give equal shares of the CPU to each thread, all three existing lock imple-

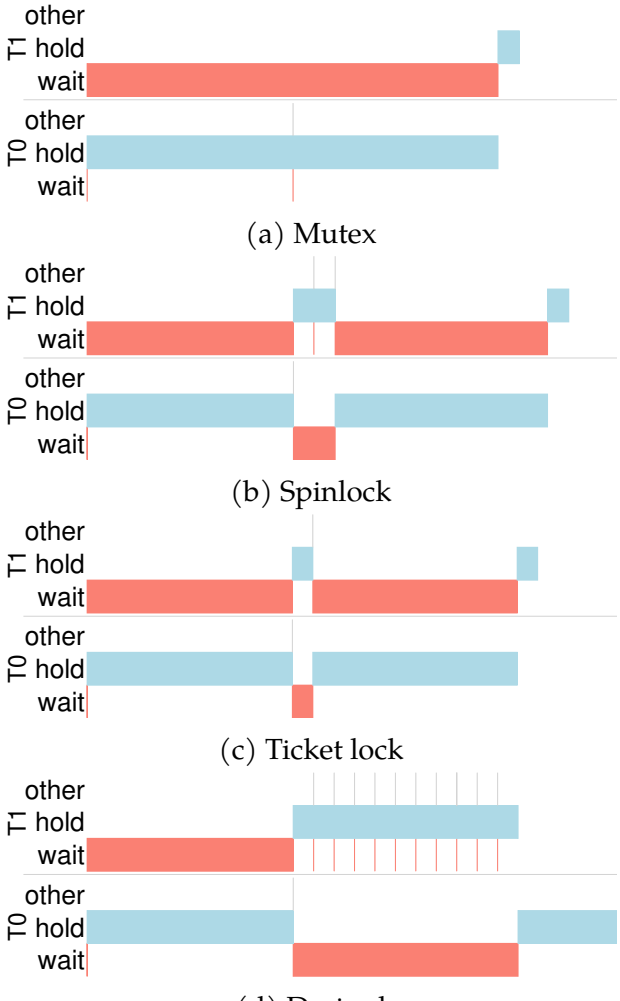


Figure 4.1: **Impact of Critical Section Size.** *The behavior of existing locks when the critical section sizes of two threads differ. The CFS scheduler is used, and each thread is pinned on a separate CPU. "wait" represents the time spent waiting to acquire the lock; "hold" represents the critical section execution, i.e., the time the lock is held; "other" represents the non-critical section execution.*

mentations enable the thread with the long critical section (T0) to obtain much more CPU.

In the case of the mutex (Figure 4.1a), T0 dominates the lock and starves T1. This behavior arises because the waiter (T1) sleeps until the lock is released. After T1 is awoken, but before getting a chance to acquire the lock, the current lock holder (T0) reacquires the mutex due to its short non-critical section time. Thus, with a mutex, one thread can dominate lock usage and hence CPU allocation.

The behavior of the spinlock (Figure 4.1b) is similar as the next lock holder is decided by the cache coherence protocol. If the current lock holder releases and reacquires the lock quickly (with a negligible non-critical section time), it can readily dominate the lock. With spinlocks, since the waiting thread busy-waits, CPU time is spent waiting without making forward progress; thus, CPU utilization will be much higher than with a mutex.

Finally, the ticket lock suffers from similar problems, even though it ensures acquisition fairness. The ticket lock (Figure 4.1c) ensures acquisition fairness by alternating which thread acquires the lock, but T0 still dominates lock usage due to its much longer critical section. Lock acquisition fairness guarantees that no thread can access a lock more than once while other threads wait; however, with varying critical section sizes, lock acquisition fairness alone cannot guarantee lock usage fairness.

This simple example shows the inability of existing locks to control the CPU allocation. One thread can dominate lock usage such that it can control CPU allocation. Additionally, for an interactive thread, when low latency matters, lock usage domination can defeat the purpose of achieving low latency goals as the thread will have to wait to acquire the lock. Thus, a new locking primitive is required, where lock usage (not just acquisition) determines when a thread can acquire a lock. The key concept of *lock opportunity* is our next focus.

	Mutex	Spinlock	Ticketlock	Desired
LOT Thread 0	20	20	20	10
LOT Thread 1	1	3	2	10
Fairness Index	0.54	0.64	0.59	1

Table 4.1: **Lock Opportunity and Fairness.** *The table shows lock opportunity and the Jain fairness index for the toy example across the range of different existing locks, as well as the desired behavior of a lock.*

4.1.2 Lock Opportunity

The non-preemptive nature of locks makes it difficult for schedulers to allocate resources when each thread may hold a lock for a different amount of time. If, instead, each thread were given a proportional "opportunity" to acquire each lock, then resources could be proportionally allocated across threads.

Lock opportunity is defined as the amount of time a thread holds a lock or *could* acquire the lock because the lock is available. Intuitively, when a lock is held, no other thread can acquire the lock, and thus no thread has the opportunity to acquire the lock; however, when a lock is idle, any thread has the opportunity to acquire the lock. The *lock opportunity time* (LOT) for thread i is formally defined as:

$$\mathrm{LOT}(i) = \sum \mathrm{Critical\_Section}(i) + \sum \mathrm{Lock\_Idle\_Time} \qquad (4.1)$$

For the toy example above, Table 4.1 shows the lock opportunity time of threads T0 and T1. We see that thread T1 has a much lower lock opportunity time than thread T0; specifically, T1 does not have the opportunity to acquire the lock while it is held by thread T0. Therefore, thread T1's

LOT is small, reflecting this unfairness.

Using lock opportunity time for each thread, we quantify the fairness using Jain's fairness index [80]; the fairness index is bounded between 0 and 1 where a 0 or 1 indicates a completely unfair or fair scenario respectively; and a score of 0.5 means the metric is fair for half of all the involved entities. As seen in the table, all three existing locks achieve fairness scores between 0.54 and 0.64, indicating that one thread dominates lock usage and thereby CPU allocation as well. Note that even though the ticket lock ensures acquisition fairness; it still has a low fairness index.

Thus, fair share allocation represents an equal *opportunity* to access each lock. For the given toy example, the desired behavior for equal lock opportunity is shown in Figure 4.1d. Once T0 acquires the lock and holds it for 10 seconds, the lock should prevent thread T0 from acquiring the lock until T1 has accumulated the same lock opportunity time. As T0 is prevented from accessing the lock, T1 has ample opportunity to acquire the lock multiple times and receive a fair allocation. Notice that at the end of 20 seconds, both threads have the same lock opportunity time and achieve a perfect fairness index of 1.

Building upon this idea, we introduce Scheduler-Cooperative Locks (SCLs). As we will see, SCLs track lock usage and accordingly adjust lock opportunity time to ensure lock usage fairness.

4.2 Scheduler-Cooperative Locks

We, now describe the goals for Scheduler-Cooperative Locks, discuss the design of SCL, and present the implementation of three types of Scheduler-Cooperative Locks: a user-space Scheduler-Cooperative Lock (u-SCL), a kernel version of u-SCL (k-SCL), and a Reader-Writer Scheduler-Cooperative Lock (RW-SCL).

4.2.1 Goals

Four high-level goals guide our SCL design:

- **Controlled lock usage allocation.** SCLs should guarantee a specified amount of lock opportunity to competing entities irrespective of their lock usage patterns. To support various scheduling goals, it should be possible to allocate different amounts of lock opportunity to different entities. Lock opportunity should be allocatable across different types of schedulable entities (e.g., threads, processes, and containers) or within an entity according to the type of work being performed.

- **High lock-acquisition fairness.** Along with lock usage fairness, SCLs should provide lock-acquisition fairness when arbitrating across threads with equal lock opportunity. This secondary criterion will help reduce wait-times and avoid starvation among active threads.

- **Minimum overhead and scalable performance.** SCLs must track the lock usage pattern of all threads that interact with a given lock, which could be costly in time and space. SCLs should minimize this overhead to provide high performance, especially with an increasing number of threads.

- **Easy to port to existing applications.** For SCLs to be widely used, incorporating SCLs into existing applications (including the OS) should be straightforward.

In this work, our primary focus is on proportional allocation. In the future, lock cooperation with other types of schedulers will be an interesting avenue of work.

4.2.2 Design

To ensure lock usage fairness without compromising performance, the design of SCL is comprised of three components.

1. **Lock usage accounting.** Each lock must track its usage across each entity that the scheduler would like to control. Accounting and classification can be performed across a wide range of entities [22]. For example, classification can be performed at a per-thread, per-process, or per-container level, or according to the type of work being performed (e.g., reading or writing). For simplicity in our discussion, we often assume that accounting is performed at the thread granularity. We believe that different types of locks can be built depending on the classification. Similarly, each thread (or schedulable entity) has a goal, or allotted, amount of lock opportunity time; this goal amount can be set to match a proportional share of the total time as desired by the CPU scheduler; for example, a default allocation would give each thread an equal fair share of lock opportunity, but any ratio can be configured.

2. **Penalizing threads depending on lock usage.** SCLs force threads that have used their lock usage quota to sleep when they prematurely try to reacquire a lock. Penalizing these threads allows other threads to acquire the lock and thus ensures appropriate lock opportunity. The potential penalty is calculated whenever a thread releases a lock and is imposed whenever a thread attempts to acquire the lock. The penalty is only imposed when a thread has reached its allotted lock usage ratio. Threads with a lower lock usage ratio than the allotted ratio are not penalized.

3. **Dedicated lock opportunity using lock slice.** Accounting for lock usage adds to lock overhead, especially for small critical sections

(lasting nanoseconds or microseconds). To avoid excessive locking overhead, we introduce the idea of a *lock slice*, building on the idea of Lock Cohorts [57]. A lock slice is similar to a time slice (or quantum) used for CPU scheduling. A lock slice is the window of time where a single thread can acquire or release the lock as often as it would like. One can view the lock slice as a *fixed size virtual critical section*. Only the lock slice owner can acquire the lock during the window. Once the lock slice expires, ownership is transferred to the next waiting thread. Thus, a lock slice guarantees lock opportunity to the thread owner; once lock ownership changes, lock opportunity changes as well. Lock slices mitigate the cost of frequent lock owner transfers and related accounting. Thus, lock slices enable usage fairness even for fine-grained acquisition patterns.

One can design many types of SCL. We now discuss the implementation of three types of SCL: u-SCL, k-SCL, and RW-SCL. While u-SCL and k-SCL guarantee lock usage fairness at a per-thread level, RW-SCL classifies threads based on the work they do (i.e., readers vs. writers).

4.2.3 u-SCL Implementation

The implementation of u-SCL is in C and is an extension of the K42 variant of the MCS lock [20]. Threads holding or waiting for the lock are chained together in a queue, guaranteeing lock acquisition. Like the K42 variant, the u-SCL lock structure also uses two pointers: tail and next. While The tail pointer points to the last waiting thread, the next pointer refers to the first waiting thread if and when there is one.

For lock accounting in u-SCL, we classify each thread as a separate class and track lock usage at a per-thread level. Per-thread tracking does incur additional memory for every active u-SCL lock. u-SCL maintains information such as lock usage, weight (which is used to determine lock

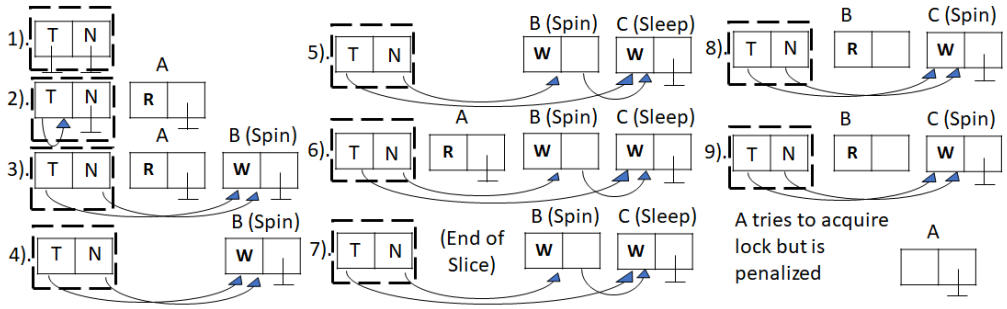


Figure 4.2: **User-space Scheduler-Cooperative Locks.** *The lock is shown as a dashed box, and each lock acquisition request is shown as a node (box). The arrow represents the pointer to the next node that forms the queue. "R" indicates running, and the lock is owned. "W" indicates waiting for the slice.*
In (1), the lock is initialized and free. (2) A single thread A has acquired the lock. The tail "T" pointer points to itself and the next "N" pointer points NULL. (3) Thread B arrives to acquire the lock and is queued. As B is the next-in-line to acquire the lock; it spins instead of parking itself. The tail and the next pointers point to B. (4) Thread A releases the lock, but B will wait for its turn as the lock slice has not expired. (5) Thread C also arrives to acquire the lock and is queued after B. C parks itself as it is not the next-in-line to acquire the lock. The tail now points to the C as it is the last one to request lock access. (6) Thread A again acquires the lock as it is the owner of the lock slice. (7) Thread A releases the lock. (8) A's lock slice is over, and B is now the owner of the slice. C is woken up and made to spin as it will acquire the lock next. The tail and the next pointers now point to C. (9) A again tries to acquire the lock but is penalized and therefore will wait for the penalty period to be over before it can be queued.

usage proportion), and penalty duration, using a per-thread structure allocated through the pthread library. The first time a thread acquires a lock, the data structure for that thread is allocated using a key associated with the lock. u-SCL does not assume a static set of threads; any number of threads can participate and can have varied lock usage patterns.

To achieve proportional allocations that match those of the CPU scheduler, u-SCL tracks the total weight of all threads and updates this information whenever a new thread requests access to a lock or the thread

exits. u-SCL identifies the nice value of the new thread and converts it to weights using the same logic that the CFS scheduler uses. The mapped weights are then added to the total weight to reflect the new proportions.

Consider two threads, T0 and T1, with nice values of 0 and -3. Based on these nice values, the CFS scheduler sets the CPU allocation ratio to approximately 1:2. u-SCL identifies the nice values and converts them to weights using the same logic that the CFS scheduler uses (0 maps to 1024 and -3 maps to 1991). The sum of the weights (1024 + 1991) is assigned to the total weight. To calculate each thread's proportion, u-SCL uses each thread's weight and the total weight to calculate the proportion. For T0 and T1, the proportion is calculated as 0.34 and 0.66, respectively, making the lock opportunity ratio approximately 1:2. This way, u-SCL guarantees lock opportunity allocations that match those of the CPU scheduler.

The interfaces *init*() and *destroy*() are used to initialize and destroy a lock respectively. The *acquire*() and *release*() routines are called by the threads to acquire and release a lock respectively. Figure 4.2 shows the operation of u-SCL. For simplicity, we just show the tail and next pointers to explain the flow in the figure and no other fields of the lock.

The figure begins with lock initialization (Step 1). If a lock is free and no thread owns the slice, a thread that tries to acquire the lock is granted access and is marked as the slice owner (Step 2). Alternatively, if the lock is actively held by a thread, any new thread that tries to acquire it will wait for its turn by joining the wait queue (Steps 3 and 5). When a lock is released (Step 4), the lock owner marks the lock as free, calculates its lock usage, and checks if the slice has expired. If the lock slice has not expired, the slice owner can acquire the lock as many times as needed (Step 6). Since a lock slice guarantees dedicated lock opportunity to a thread, within a lock slice, lock acquisition is fast-pathed, significantly reducing lock overhead.

On the other hand, if the lock slice has expired, the current slice owner

sets the next waiting thread to be the slice owner (Steps 7 and 8). The current slice owner also checks to see if it has exceeded the lock usage ratio to determine an appropriate penalty. When acquiring the lock, another check is performed by the requesting thread to see if it should be penalized for overusing the lock earlier. A thread whose lock usage ratio is above the desired ratio is banned until other active threads are given sufficient lock opportunity. The thread is banned by forcing it to sleep (Step 9) and can try to acquire the lock after the penalty is imposed.

We have chosen two milliseconds as the slice duration for all experiments unless otherwise stated. The two-millisecond duration optimizes for throughput at the cost of longer tail latency. We will show the impact of slice duration on throughput and latency in Section 4.3.4.

Optimizations. To make u-SCL efficient, we use the spin-and-park strategy whenever the thread does not immediately acquire the lock; since waiting threads sleep the majority of the time, the CPU time spent while waiting is minimal.

Another optimization we implement is *next-thread prefetch* where the waiting thread that will next acquire the lock is allowed to start spinning (Steps 3 and 8); this mechanism improves performance by enabling a fast switch when the current lock holder is finished with its slice. When this optimization marks the next waiting thread as runnable, the scheduler will allocate the CPU. However, as the thread has not acquired the lock, it will spin, wasting the CPU. This behavior will be more visible when the number of threads is greater than the number of cores available.

Limitations. We now discuss a few of the implementation limitations. First, threads that sporadically acquire a lock continue to be counted in the total weight of threads for that lock; hence, other active threads may receive a smaller CPU allocation. This limitation is addressed in our kernel implementation of k-SCL.

Next, our current implementation of u-SCL does not update lock

weights when the scheduler changes its goals (e.g., when an administrator changes the nice value of a process); this is not a fundamental limitation. Finally, we have not yet explored u-SCL in multi-lock situations (e.g., with data structures that require hierarchical locking or more complex locking relationships). We anticipate that multiple locks can interfere with the fairness goals of each lock leading to performance degradation. The interaction of multiple SCLs remains as future work.

Lastly, currently, SCLs incur high space overhead as they have to maintain the accounting information for each thread. With hundreds of SCLs used in an application, the total space overhead will be significant and may lead to performance implications.

4.2.4 k-SCL Implementation

k-SCL is the kernel implementation of u-SCL; it is a much simpler version without any optimizations discussed above. As larger lock slice sizes increase the wait-time to own the lock slice, we set the lock slice size to zero while accessing kernel services. Unlike u-SCL, which uses a per-thread structure to store the accounting information, k-SCL uses a hash table to store and retrieve the accounting information for each thread.

k-SCL uses the idea of ticket locks to ensure lock acquisition fairness. If a thread is acquiring the lock for the first time, it will register itself for future lock acquisition. Otherwise, the thread will check if it needs to be penalized for the previous lock acquisition or not. If it needs to be penalized, then the thread waits until the penalty time expires. After the penalty check, the thread acquires the next ticket and waits for its turn to acquire the lock. Once the thread is ready to enter the critical section, it simply notes the current time as the start of the critical section for accounting purposes.

After the critical section is executed, while releasing the lock, the thread calculates the critical section duration and then, based on it, cal-

culates the penalty. The thread remembers the time until when it cannot acquire the lock again. Lastly, the thread increases the ticket by one to let another waiting thread acquire the lock.

Check active interaction. The primary difference between k-SCL and u-SCL is that k-SCL *does* track whether or not threads are actively interacting with a kernel lock and accordingly removes them from the accounting. This check is performed by periodically walking through the list of all thread data and freeing inactive threads data. Inactive threads are threads that will either never acquire the lock again or will spend a long time doing something else before acquiring the lock again (i.e., have a high non-critical-section time). All other threads are considered active by k-SCL.

It is challenging to accurately detect the inactive threads as k-SCL cannot determine if the thread has exited or will not acquire the lock again in the future. Therefore, k-SCL relies on heuristics to detect inactive threads. To detect inactive threads, either a centralized or decentralized option is present.

In the centralized option, a separate thread can periodically scan all threads and tag the ones that did not request a lock acquisition in the last period as inactive. However, we decide not to use this centralized approach as a separate thread per lock is needed to monitor the inactive threads. This approach would have trouble scaling as thousands of such monitoring threads are needed for thousands of locks used by an application or operating system.

Instead, we choose a decentralized approach, whereupon the lock release, the thread checks all the other threads that arrived before to find the first active thread. All the threads in between are freed to reduce the total thread count. Currently, we use a threshold to determine if a thread is inactive or not. The threshold can either be set statically or can be learned over time. In our implementation, we set the threshold value of one sec-

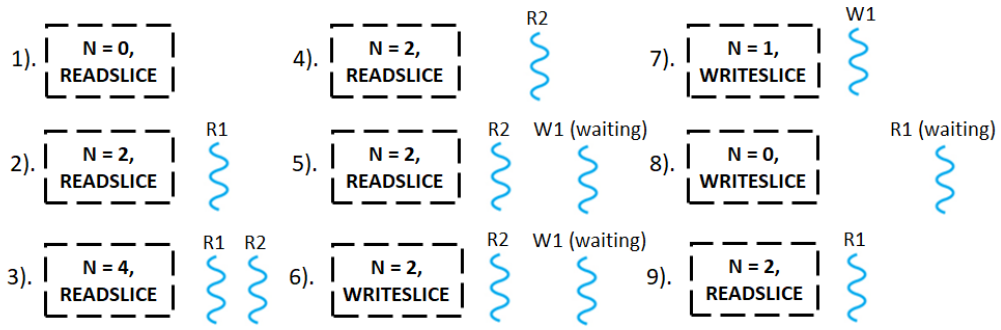


Figure 4.3: **Reader-Writer Scheduler-Cooperative Locks.** *The dashed box represents the lock. "N" represents the value of the counter. "READSLICE" and "WRITESLICE" represents the read and write slice that the lock is currently in. In (1) The lock is initialized and free. (2) A reader thread R1 acquires the lock and continues its execution in the critical section. (3) Another reader thread, R2, also joins in. (4) Reader R1 leaves the critical section. (5) A writer thread W1 arrives and waits for the write slice to start. (6) Write slice is started, and W1 waits for reader R2 to release the lock. (7) Reader R2 releases the lock and the writer W1 acquires the lock. (8) Reader R1 now arrives again and waits for the read slice to start. (9) W1 releases the lock, and the read slice starts allowing the reader R1 to acquire the lock.*

ond. A longer threshold can lead to stale accounting for a longer duration, leading to performance issues. The drawback with such an approach is that threads need to access other threads' information.

4.2.5 RW-SCL Implementation

RW-SCL provides the flexibility to assign a lock usage ratio to readers and writers, unlike the existing reader-writer locks that support reader or writer preference. The implementation of RW-SCL is in C and is an extension of the centralized reader-writer lock described by Scott [138]. Being centralized, RW-SCL tracks the number of readers and writers with a single counter; the lowest bit of the counter indicates if a writer is active,

while the upper bits indicate the number of readers that are either active or are waiting to acquire the lock. To avoid a performance collapse due to heavy contention on the counter, we borrow the idea of splitting the counter into multiple counters, one per NUMA node [34].

With RW-SCL, threads are classified based on the type of work each executes; the threads that execute read-only operations belong to the reader class, while the threads executing write operations belong to the writer class. Since there are only two classes, RW-SCL does not use per-thread storage and track each thread. Fairness guarantees are provided across the set of reader threads and the set of writer threads. As with other SCL locks, different proportional shares of the lock can be given to readers versus writers. For the current implementation, we assume that all the readers will have the same priority. Similarly, all writers will have the same priority. Thus, a single thread cannot assume the role of a reader and writer as the same nice value will be used, leading to a 50:50 usage proportion, which might not be the desired proportion.

Figure 4.3 shows the operation of RW-SCL. The relevant RW-SCL routines include *init*(), *destroy*(), *writer_lock*(), *reader_lock*(), *reader_unlock*(), and *reader_unlock*(). The lock begins in a read slice at initialization (Step 1). During a read slice, the readers that acquire the lock atomically increments the counter by two (Step 2 and 3). On releasing the lock, the counter is atomically decremented by two (Step 4). During the read slice, all the writers that try to acquire the lock must wait for the write slice to be active (Step 5). When readers release the lock, they will check if the read slice has expired and may activate the write slice (Step 6).

While the write slice is active, writers try to acquire the lock by setting the lowest bit of the counter to 1 using the compare-and-swap instruction (Step 7). With multiple writers, only one writer can succeed, and other writers must wait for the first writer to release the lock. If a reader tries to acquire the lock while the write slice is active, it will wait for the read

slice to be active (Step 8). When a writer releases the lock, it will check if the write slice has expired and activate the read slice (Step 9). Whenever a read slice changes to a write slice or vice versa, the readers and writers will wait for the other class of threads to drain before acquiring the lock.

As RW-SCL does not track per-thread lock usage, RW-SCL cannot guarantee writer-writer fairness; however, RW-SCL can improve performance for multiple writers. Within the write class, multiple writers can contend for the lock, and thus while one writer is executing the non-critical section, another writer can acquire the lock and execute. This behavior is contrary to the u-SCL behavior, where the lock remains unused when the owner is executing the non-critical section code.

4.3 Evaluation

In this section, we evaluate the effectiveness of SCLs. Using microbenchmarks, we show that u-SCL provides both scalable performance and scheduling control. We also show how SCLs can be used to solve real-world problems by replacing the existing locks in UpScaleDB with u-SCL, the Linux rename lock with k-SCL, and the existing reader-writer lock in KyotoCabinet with RW-SCL.

We perform our experiments on a 2.4 GHz Intel Xeon E5-2630 v3. It has two sockets; each socket has eight physical cores with hyper-threading enabled. The machine has 128 GB RAM and one 480 GB SAS SSD. The machine runs Ubuntu 16.04 with kernel version 4.19.80, using the CFS scheduler.

We begin with a synthetic workload to stress different aspects of traditional locks as well as u-SCL. The workload consists of a multi-threaded program; each thread executes a loop and runs for a specified amount of time. Each loop iteration consists of two elements: time spent outside a shared lock, i.e., non-critical section, and time spent with the lock ac-

quired, i.e., critical section. Both are specified as parameters at the start of the program. Unless explicitly specified, the priority of all the threads is the default, thereby ensuring each thread gets an equal share of the CPU time according to the CFS default policy.

We use the following metrics to show our results:

1. **Throughput:** For synthetic workloads, throughput is the number of times through each loop, whereas, for real workloads, it reflects the number of operations completed (e.g., inserts or deletes). This metric shows the bulk efficiency of the approach.

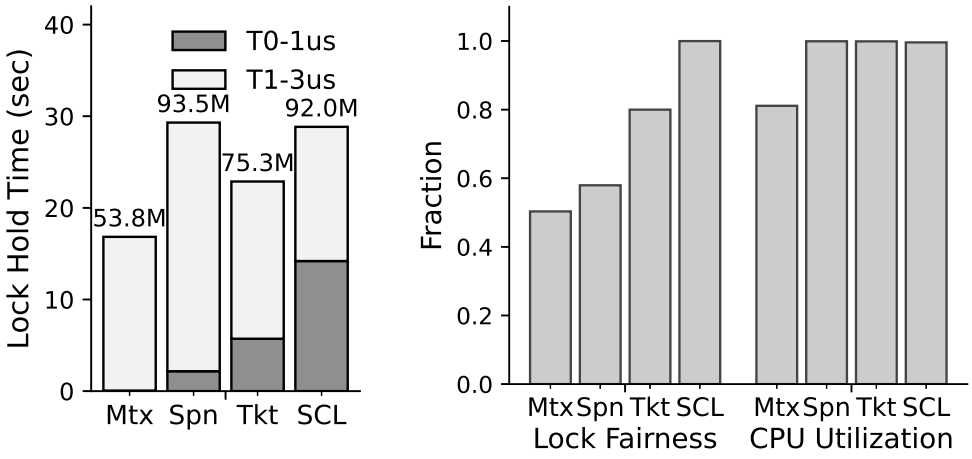
2. **Lock Hold Time:** This metric shows the time spent holding the lock, broken down per thread. This shows whether the lock is being shared fairly.

3. **Lock Usage Fairness:** This metric captures fair lock usage among all threads. We use the method described in Section 4.1 to calculate lock opportunity time.

4. **CPU Utilization:** This metric captures how much total CPU is utilized by all the threads to execute the workload. CPU utilization ranges from 0 to 1. A higher CPU utilization means that the threads spin to acquire the lock, while a lower CPU utilization means the lock may be more efficient due to blocking. Lower CPU utilization is usually the desired goal.

4.3.1 Fairness and Performance

To gauge the fairness and performance of u-SCL compared to traditional locks, we first run a simple synthetic workload with two threads. For this 30 second workload, the critical section sizes are 1 µs and 3 µs, and the two threads are pinned on two different CPUs. In these experiments, the desired result is that for fair scheduling, each thread will hold the lock



(a) LHT and Throughput (b) Lock usage fairness and CPU utilization

Figure 4.4: **Comparison on 2 CPUs.** *The graphs present a comparison of four locks: mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for 2 (a and b) and 16 (c and d) threads, each has the same thread priority. For 2 threads, each thread has a different critical section size (1 µs vs. 3 µs). For 16 threads, half have shorter critical section sizes (1 µs) while others have a larger critical section size (3 µs). "TG" stands for the thread group.*

for the same amount of time, and for performance, they will complete as many iterations as possible.

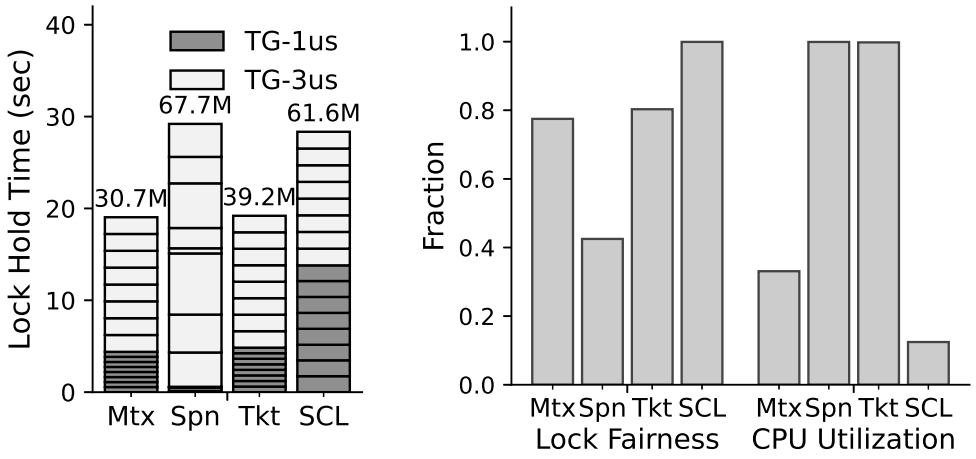
Figure 4.4a shows the amount of time each of the two threads holds the lock (dark for the thread with a 1 µs critical section, light for 3 µs); the top of each bar reports throughput. The mutex, spinlock, and ticket lock each do not achieve equal lock hold times. For the mutex, the thread with the longer (3 µs) critical section (light color) is almost always able to grab the lock and then dominate usage. With the spinlock, behavior varies from run to run, but often, as shown, the thread with the longer critical section dominates. Finally, with the ticket lock, threads hold the lock in direct proportion to lock usage times, and thus the (light color) thread with the 3 µs critical section receives three-quarters of the lock

hold time. In contrast, u-SCL apportions lock hold time equally to each thread. Thus, u-SCL achieves one of its most important goals. Figure 4.4b summarizes these results with Jain's fairness metric for lock hold time. As expected, u-SCL's lock usage fairness is 1 while that of other locks is less than 1.

Figure 4.4a also shows overall throughput (the numbers along the top). As one can see, u-SCL and the spinlock are the highest performing. The mutex is slowest, with only 53.8M iterations; the mutex often blocks the waiting thread using futex calls, and the thread switches between user and kernel mode quite often, lowering performance. For CPU utilization, the mutex performs better than others since the mutex lets the waiters sleep by calling *futex_wait*(). On the other hand, the spinlock and ticket lock spin to acquire the lock, and thus their utilization is high. u-SCL's high CPU utilization is attributed to the implementation decision of letting the next thread (that is about to get the lock) spin. However, with more threads, u-SCL is extremely efficient, as seen next.

We next show how u-SCL scales by running the same workload as above with 16 threads on 16 cores. For this experiment, the critical section size for half of the threads (8 total) is 1 µs, and another half (also 8) is 3 µs, respectively. From Figure 4.5a, we see again that the three traditional locks – mutex, spinlock, and ticket lock – are not fair in terms of lock usage; not all threads have the same lock hold times. The threads with larger critical section sizes (lighter color) dominate the threads with shorter critical section sizes (darker). In contrast, u-SCL ensures that all threads receive the same lock opportunity irrespective of critical section size. The throughput with u-SCL is comparable to that of spinlock, and we believe that further tuning could reduce this small gap.

However, the more important result is found in Figure 4.5b, which shows CPU utilization. u-SCL's CPU utilization is reduced significantly compared to other spinning (spinlock and ticket lock) approaches. With



(a) LHT and Throughput (b) Lock usage fairness and CPU utilization

Figure 4.5: **Comparison on 16 CPUs.** *The graphs present a comparison of four locks: mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for 2 (a and b) and 16 (c and d) threads, each has the same thread priority. For 2 threads, each thread has a different critical section size (1 µs vs. 3 µs). For 16 threads, half have shorter critical section sizes (1 µs) while others have a larger critical section size (3 µs). "TG" stands for the thread group.*

u-SCL, out of 16 threads, only two are actively running at any instance, while all other threads are blocked. u-SCL carefully orchestrates which threads are awake and which are sleeping to minimize CPU utilization while also achieving fairness. While a mutex conserves CPU cycles, it has a much higher lock overhead, delivers much lower throughput, and does not achieve fairness.

In summary, we show that u-SCL provides lock opportunity to all threads and minimizes the effect of lock domination by a single thread or a group of threads, thus helping to avoid the scheduler subversion problem. While ensuring the fair-share scheduling goal, u-SCL also delivers high throughput and very low CPU utilization. u-SCL thus nearly combines all the good qualities of the three traditional locks – the perfor-

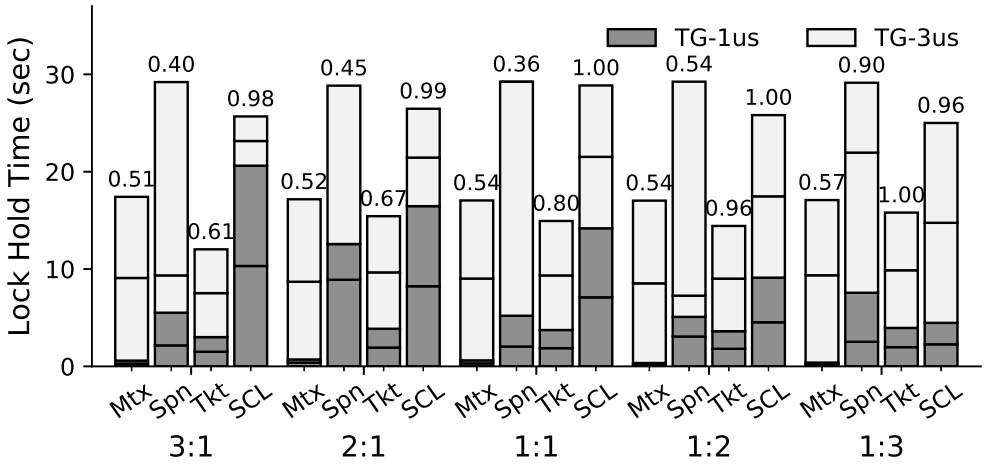


Figure 4.6: **Changing Thread Proportionality.** *Comparison of the four locks: mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for four threads running on two CPUs having different thread priorities (shown as ratios along the bottom) and different critical section sizes. The number on the top of each bar shows the lock usage fairness.*

mance of spinlock, the acquisition fairness of the ticket lock, and the low CPU utilization of the mutex.

4.3.2 Proportional Allocation

We next demonstrate that u-SCL enables schedulers to proportionately schedule threads according to a desired ratio other than 50:50. Figure 4.6 shows the performance of all four locks when the desired CPU time allocation is varied. We now consider four threads pinned to two CPUs, while the other workload parameters remain the same (the critical sections for two threads are 1 µs, for the other two threads, they are 3 µs; the workload runs for 30 seconds).

To achieve different CPU proportions for the thread groups, we vary the CFS *nice* values. The leftmost group (3:1) indicates that shorter critical section threads (darker color) should receive three times the CPU of

longer critical section threads (lighter). The rightmost group shows the inverse ratio, with the shorter critical section threads meant to receive one third the CPU of the longer critical section threads.

Figure 4.6 shows that traditional locks subvert the target CPU allocations of the CFS scheduler. Having a longer critical section leads to threads holding onto the CPU for a longer duration. Note that the fairness of ticket locks improves whenever the critical section ratio and the thread priority matches.

In contrast, u-SCL performs exactly in alignment with CPU scheduling goals and allocates the lock in the desired proportion to each thread. To do so, u-SCL uses the same weight for the lock usage ratio as the thread's scheduling weight, thereby guaranteeing CPU and lock usage fairness. Thus, by configuring u-SCL to align with scheduling goals, the desired proportional-sharing goals are achieved.

4.3.3 Lock Overhead

Minimal overhead was one of our goals while designing u-SCL. To understand the overhead of u-SCL, we conduct two different experiments as we increase the number of threads with very small critical sections.

For the first experiment, we set each critical section and non-critical section size to 0. We then run the synthetic application varying the number of threads from 2 to 32, with each thread pinned to a CPU core.

Figure 4.7a (left) compares the throughput of the four lock types. For spinlocks and ticket locks, throughput is generally low and decreases as the number of threads increases; these locks generate a great deal of cache coherence traffic since all threads spin while waiting. In contrast, the mutex and u-SCL block while waiting to acquire a lock and hence perform better. While u-SCL significantly outperforms the other locks for 8 or fewer threads, u-SCL performance does drop for 16 and 32 threads when the threads are running on different NUMA nodes; in this config-

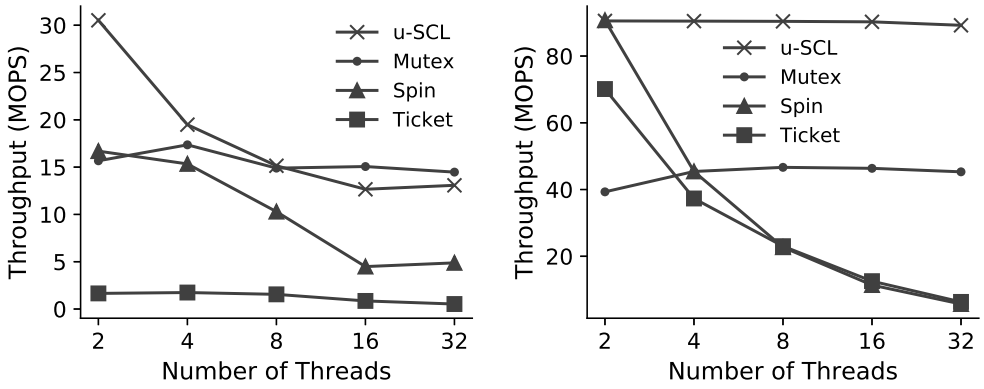


Figure 4.7: **Lock Overhead Study.** *The figure presents two lock overhead studies. On the left(a), the number of threads and CPU cores are increased, from 2 to 32, to study scaling properties of u-SCL and related locks. We pin each thread on a separate CPU. On the right(b), the number of CPUs is fixed at two, but the number of threads is increased from 2 to 32.*

uration, there is an additional cost to maintain accounting information due to cross-node cache coherency traffic. This indicates that u-SCL could benefit from approximate accounting across cores in future work.

In the second experiment, we show performance when the number of CPUs remains constant at two, but the number of threads increases. For the experiment in Figure 4.7a (right), we vary the number of threads from 2 to 32 but pin them to only two CPUs. The critical section size is 1 µs.

As expected, the performance of u-SCL and mutex is better than the alternatives and remains almost constant as the number of threads increases. With spinlock and ticket locks, the CPU must schedule all the spin-waiting threads on two CPUs, even though threads cannot make forward progress when they are not holding the lock. Moreover, as the threads never yield the CPU until the CPU time slice expires, a great deal of CPU time is wasted.

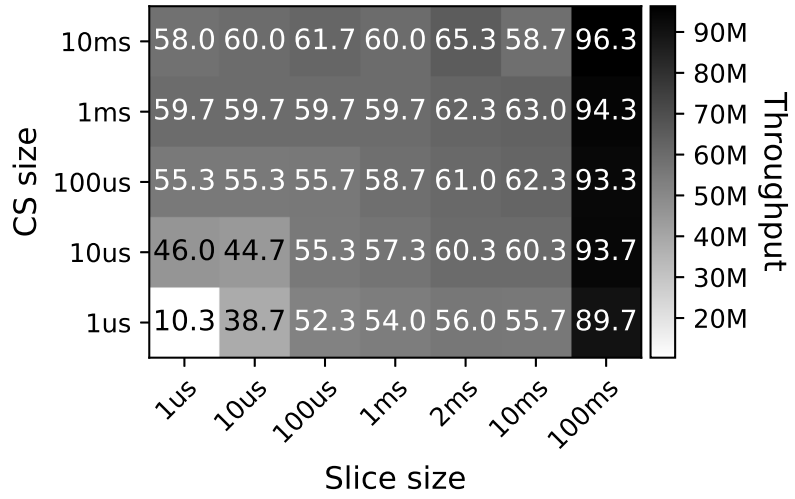
In contrast, with u-SCL and mutex, only two threads or one thread, respectively, are running, which significantly reduces CPU scheduling. u-SCL performs better than mutex since the next thread to acquire the lock is effectively prefetched; the dedicated lock slice also helps u-SCL achieve higher performance since lock overhead is minimal within a lock slice. With a mutex lock, a waiting thread must often switch between user and kernel mode, lowering performance.

4.3.4 Lock Slice Sizes vs. Performance

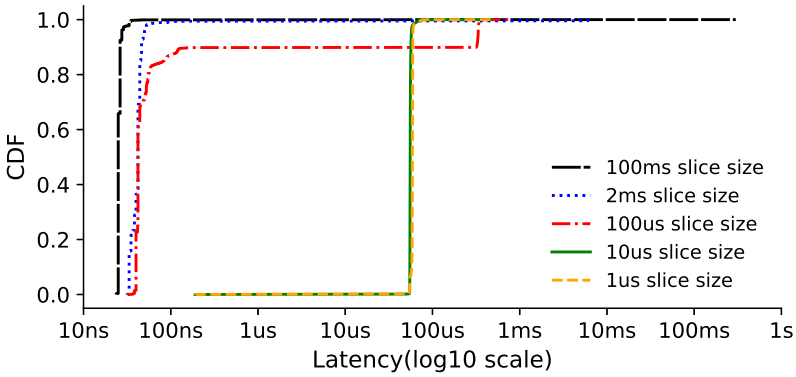
We next show the impact of lock slice size on throughput and latency as a function of critical section size. In general, increasing lock slice size increases throughput but harms latency. As we will show, the default two-millisecond slice size optimizes for high throughput at the cost of long-tail latency.

Throughput: For our first workload, we run four identical threads pinned to two cores for 30 seconds, varying the size of each critical section. Figure 4.8a shows throughput in a heatmap. The x-axis varies the lock slice size while the y-axis varies the critical section size. Throughput is calculated by summing the individual throughput of each thread. For larger slice sizes, the throughput increases, while for very small slice sizes, the throughput decreases significantly; the overhead of repeatedly acquiring and releasing the lock causes a substantial decrease.

Latency: Figure 4.8b shows the wait-time distribution to acquire the lock as a function of the slice size for a 10 µs critical section; we chose 10 µs to show the impact of having a slice size smaller, greater, or equal to the critical section size. We omit 1 ms and 10 ms lock slices because those results are similar to that of 2 ms. The figure shows that for lock slices larger than the critical section, the wait-time of the majority of operations is less than 100 ns; each thread enters the critical section numerous times without any contention. However, when the thread does not own the lock



(a) Heatmap showing throughput performance.



(b) Latency wait-time distribution.

Figure 4.8: **Impact of lock slice size on performance.** *The top figure (a) shows the throughput across two dimensions: critical section and the slice size. The bottom figure (b) shows the wait-time distribution when the lock slice varies, and the critical section size is 10 µs.*

slice; it must wait for its turn to enter the critical section; in these cases, the wait time increases to a value that depends on the lock slice size and the number of participating threads.

When the lock size is smaller than or equal to the size of the critical

section (i.e., 1 or 10 µs), lock ownership switches between threads after each critical section, and each thread observes the same latency. We observe this same relationship when we vary the size of the critical section (not shown).

To summarize, with larger lock slices, throughput increases but a small portion of operations observe high latency, increasing tail latency. On the other hand, with smaller lock slices, latency is relatively low, but throughput decreases tremendously. Hence, latency-sensitive applications should opt for smaller lock slices, while applications that need throughput should opt for larger lock slices.

Interactive jobs: We now show that u-SCL can deliver low latency to interactive threads in the presence of batch threads. Batch threads usually run without user interaction and thus do not require low scheduling latency [27]. On the other hand, interactive threads require low scheduling latency; thus, the scheduler's task is to reduce the wait-time for interactive threads so they can complete tasks quickly. Both Linux's CFS and FreeBSD's ULE schedulers identify interactive and batch threads and schedule them accordingly [134]; for example, as the interactive threads sleep more often without using their entire CPU slice, CFS schedules such threads before others to minimize their latency.

To show that u-SCL can effectively handle both batch and interactive threads, we examine a workload with one batch thread and three interactive threads. The batch thread repeatedly acquires the lock, executes a 100 µs critical section, and releases the lock; the three interactive threads execute a 10 µs critical section, release the lock and then sleep for 100 µs. The four threads are pinned on two CPUs. The desired result is that the interactive threads should not need to wait to acquire the lock.

Figure 4.9 shows the CDF of the wait-time for one of the interactive threads to acquire each of the four lock types: mutex, spinlock, ticket lock, and u-SCL. For u-SCL, we show four lock slice sizes. The results

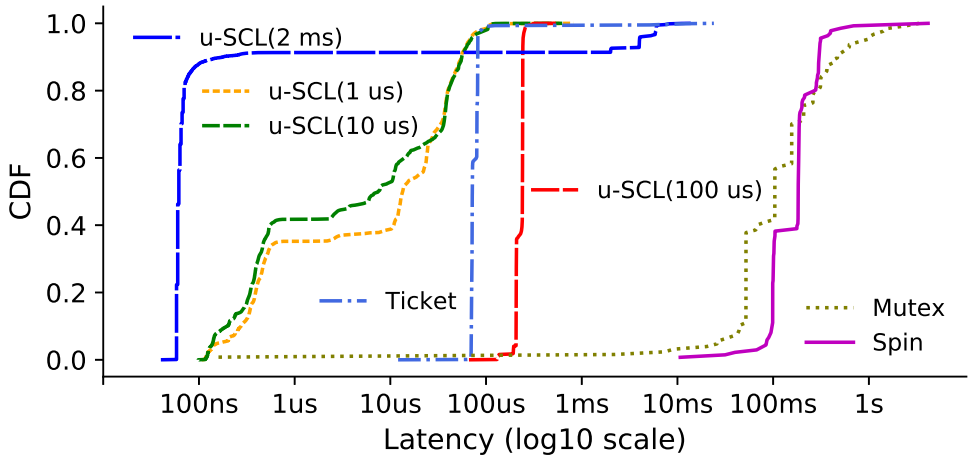


Figure 4.9: **Interactivity vs. Batching.** *The figure shows the comparison of the wait-time to acquire the lock for mutex, spinlock, ticket lock and u-SCL.*

of the other interactive threads are similar. The graph shows that for the mutex and spinlock, wait-time is very high, usually between 10 ms and 1 second. Even though the goal of the scheduler is to reduce latency by scheduling the interactive thread as soon as it is ready, lock ownership is dominated by the batch thread, which leads to longer latency for the interactive threads. The ticket lock reduces latency since lock ownership alternates across threads; however, wait-time is still high because the interactive thread must always wait for the critical section of the batch thread to complete (100 µs).

The graph shows that for u-SCL, the length of the slice size has a large impact on wait-time. When the slice size is smaller than or equal to the interactive thread's critical section (e.g., 1 or 10 µs), the interactive thread often has a relatively short wait time and never waits longer than the critical section of the batch thread (100 µs). When the slice size is relatively large (e.g., 2 ms), the interactive thread often acquires the lock with no waiting, but the wait-time distribution has a long tail. Finally, the 100

µs slice performs the worst of the u-SCL variants because the interactive threads sleep after releasing the lock, wasting the majority of the lock slice and not allowing other waiting threads to acquire the lock.

In summary, to deliver low latency, the slice size in u-SCL should always be less than or equal to the smallest critical section size. Our initial results with the ULE scheduler are similar, but a complete analysis remains as future work.

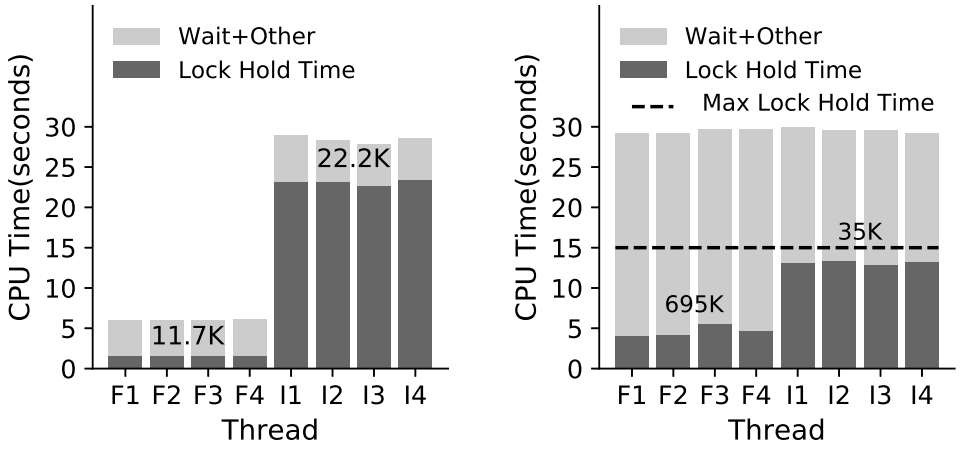
4.3.5 Real-world Workloads

We conclude our investigation by demonstrating how SCLs can be used to solve real-world scheduling subversion problems. We concentrate on two applications, UpScaleDB and KyotoCabinet, and a shared lock within the Linux kernel. The user-space applications show how SCLs can avoid the scheduler subversion problem within a single process. With the kernel example, we illustrate a competitive environment scenario where multiple applications running as different processes (or containers) can contend for a lock within a kernel, thus leading to cross-process scheduler subversion.

4.3.5.1 UpScaleDB

As part of the original motivation for u-SCL, we saw in Figure 3.1 that UpScaleDB was unable to deliver a fair share of the CPU to threads performing different operations. For easy comparison, Figure 4.10a shows the same graph. We now show that u-SCL easily solves this problem; the existing locks in UpScaleDB can simply be converted to u-SCL locks and then the unmodified Linux CFS scheduler can effectively schedule UpScaleDB's threads independent of their locking behavior.

We repeat the experiment shown in Figure 3.1, but with u-SCL locks.



(a) Mutex (b) u-SCL

Figure 4.10: **Mutex and u-SCL performance with UpScaleDB.** *The same workload is used as Section 3.1.1. The same CFS scheduler is used for the experiments. "F" denotes find threads while "I" denotes insert threads. The expected maximum lock hold time is shown using the dashed line. "Hold" represents the critical section execution, i.e., the time until the lock is held; "Wait + Other" represents the wait-times and non-critical section execution. The number on top of the dark bar represents the throughput (operations/second). The left figure (a) shows the same graph as shown in Section 3.1.1. The right figure (b) shows the performance of u-SCL.*

UpScaleDB is configured to run a standard benchmark.[1] We again consider four threads performing find operations and four threads performing inserts pinned to four CPUs. Figure 4.10b shows how much CPU time each thread is allocated by the CFS scheduler.

As desired, with u-SCL, the CPU time allocated to each type of thread (i.e., threads performing find or insert operations) corresponds to a fair share of the CPU resources. With u-SCL, all threads, regardless of the operation they perform (or the duration of their critical section), are sched-

[1] ups_bench –use-fsync –distribution=random –keysize-fixed –journal-compression=none –stop-seconds=120 –num-threads=N

uled for approximately the same amount of time: 30 seconds. In contrast, with mutexes, UpScaleDB allocated approximately only 2 CPU seconds to the find threads and 24 CPU seconds to the insert threads (with four CPUs). With a fair amount of CPU scheduling time, the lock hold times become fairer as well; note that it is not expected that each thread will hold the lock for the same amount of time since each thread spends a different amount of time in critical section versus non-critical section code for its 30 seconds.

The graph also shows that the overall throughput of find and insert threads is greatly improved with u-SCL compared to mutexes. On four CPUs, the throughput of find threads increases from only about 12K ops/sec to nearly 700K ops/sec; the throughput of insert threads also increases from 22K ops/sec to 35K ops/sec. The throughput of find operations increases dramatically because find threads are now provided more CPU time and lock opportunity. Even the throughput of inserts improves since u-SCL provides a dedicated lock slice where a thread can acquire the lock as many times possible; thus, lock overhead is greatly reduced.

Finally, when we sum up the total lock utilization across the u-SCL and mutex versions, we find that the lock is utilized for roughly 59% of the total experiment duration for u-SCL but nearly 83% for mutexes. Given that the overall lock utilization decreases with increased throughput, we believe that u-SCL can help scale applications. Therefore, instead of redesigning the applications to scale by minimizing critical section length, a more scalable lock can be used.

4.3.5.2 KyotoCabinet

KyotoCabinet [93] is an embedded key-value storage engine that relies on reader-writer locks. Given that locks are held for significant periods of time and critical sections are of different lengths, it also suffers from scheduler subversion; specifically, writers can be easily starved. We show

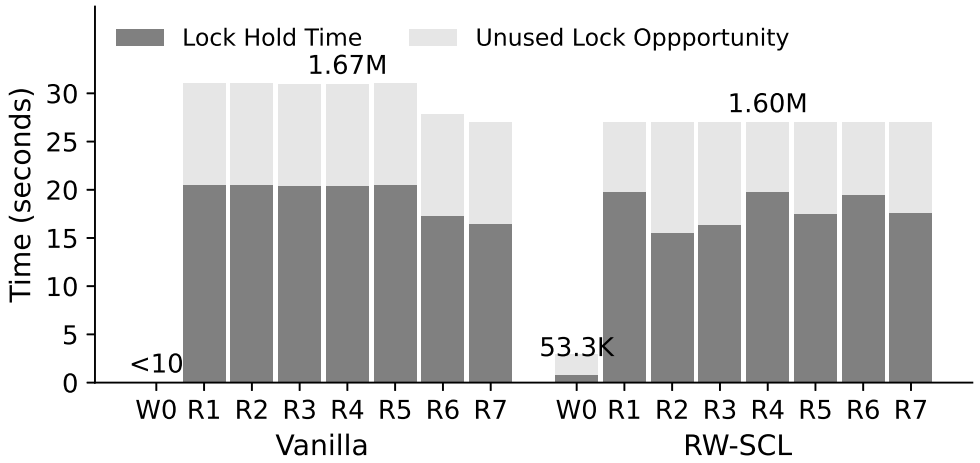


Figure 4.11: **Comparison of RW-SCL and KyotoCabinet** *The dark bar shows the lock hold time for each individual thread and the light bar shows the lock opportunity not being unused. The values on top of the bar shows the aggregated throughput (operations/sec) for the writer and reader threads.*

that our implementation of RW-SCL allows KyotoCabinet and the default CFS Linux scheduler to control the amount of CPU resources given to readers and writers, while still delivering high throughput; specifically, RW-SCL removes the problem of writers being starved.

To setup this workload, we use KyotoCabinet's built-in benchmarking tool kccachetest in wicked mode on an in-memory hash-based database. We modified the tool to let the thread either issue read-only (reader) or write (writer) operations and run the workload for 30 seconds. The database contains ten million entries that are accessed at random. We pin threads to cores for all the experiments. We assign a ratio of 9:1 to the reader and writer threads. The original version of KyotoCabinet uses pthread reader-writer locks.

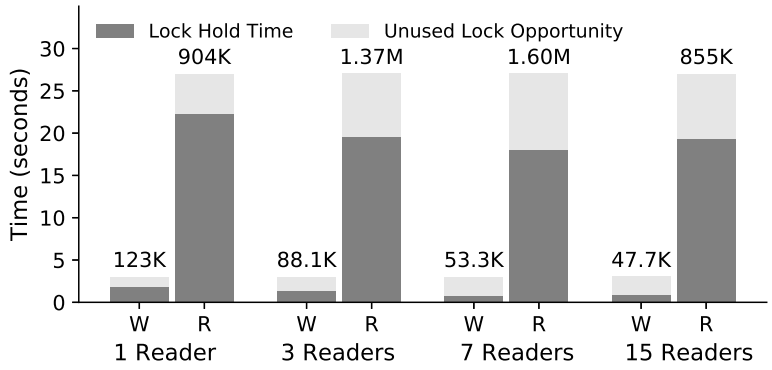
To begin, we construct a workload with one writer thread and seven reader threads. In Figure 4.11, we present the write throughput and the aggregated read throughput, the average lock hold time, and the lock

opportunity for readers and writers. The default KyotoCabinet using pthread reader-writer lock gives strict priority to readers, the writer is starved, and less than ten write operations are performed over the entire experiment. In other experiments (not shown), we find that the writer starves irrespective of the number of readers.

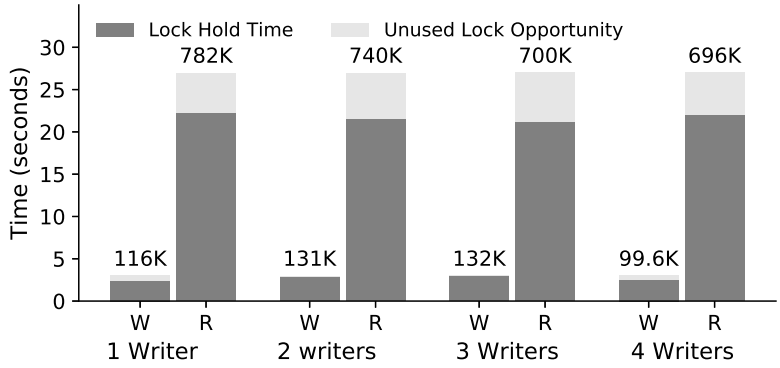
On the other hand, RW-SCL ensures that the writer thread obtains 10% of the lock opportunity compared to 90% for the readers (since readers can share the reader lock, their lock opportunity time is precisely shared). Since the writer thread is presented with more lock opportunity with RW-SCL, the write throughput increases significantly compared to the vanilla version. As expected, RW-SCL read throughput decreases slightly because the reader threads now execute for only 90% of the time, and writes lead to many cache invalidations. Nevertheless, the overall aggregated throughput of readers and writers for eight threads with RW-SCL is comparable to other reader-writer locks with KyotoCabinet [55].

We run similar experiments by varying the number of readers and show the result in Figure 4.12a. KyotoCabinet scales well until 8 threads (7 readers + 1 writer). The throughput drops once the number of threads crosses a single NUMA node. We believe that this performance drop is due to the excessive data sharing of KyotoCabinet data structure across the sockets. Another point to note here is that irrespective of the number of readers, RW-SCL continues to stick to the 9:1 ratio that we specified.

When there is only one writer thread with RW-SCL, the writer cannot utilize its entire write slice since the lock is unused when the writer is executing non-critical section code. To show how multiple writers can utilize the write slice effectively; we conduct another experiment with only one reader while varying the number of writers. As seen in Figure 4.12b, when the number of writers increases from one to two, the lock opportunity time becomes completely used as lock hold time (as desired); when one writer thread is executing the non-critical section code,



(a) Reader Scaling



(b) Writer Scaling

Figure 4.12: **Performance of RW-SCL with reader and writer scaling.** *For reader scaling, only one writer is used while for writer scaling, only one reader is used. The number of readers and writers vary for reader scaling and writer scaling experiments. The dark bar shows the lock hold time while the light bar shows the unused lock opportunity. The values on top of the bar shows the throughput of the writers and readers.*

the other writer thread can acquire the lock, thereby fully utilizing the write slice. Increasing the number of writers past two cannot further increase write lock hold time or therefore improve throughput; continuing to increase the number of writers past three simply increases the amount of cache coherence traffic.

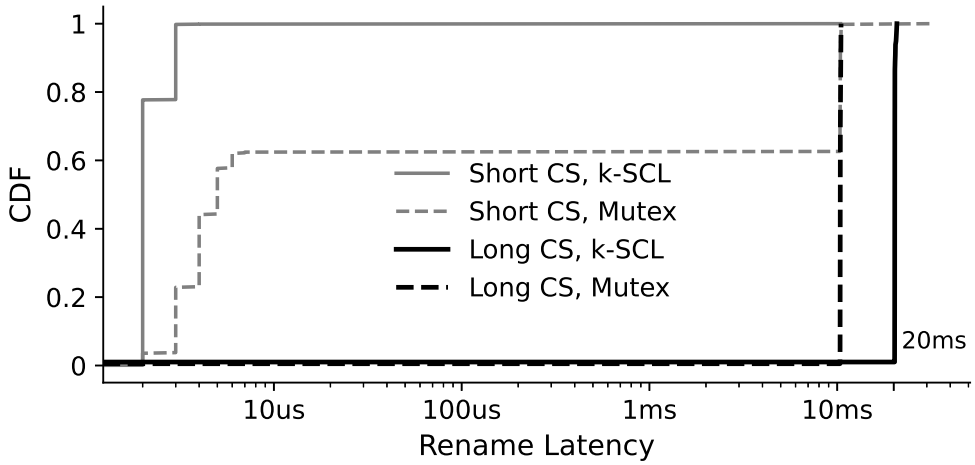


Figure 4.13: **Rename Latency.** *The graph shows the latency CDFs for SCL and the mutex lock under the rename operation. The dark lines show the distributions for the long rename operation (the bully), whereas lighter lines represent the short rename operation costs (the victim). Dashed lines show standard mutex performance, whereas solid lines show k-SCL performance.*

4.3.5.3 Linux Rename Lock

A cross-directory rename in Linux is a complicated operation that requires holding a global mutex to avoid a deadlock. When accessed by competing entities, such a global lock can lead to performance problems since all threads needing to perform a rename must wait to acquire it. Using a bully and victim process, we describe an input parameter attack. Then, we show that k-SCL can prevent a bully process that holds the global lock for long periods of time from starving out other processes that must also acquire the global lock.

Our specific experiment to recreate a bully and a victim process is as follows. We use Linux version 4.9.128 and the ext4 file system; we have disabled dir_index using tune2fs [35] since that optimization leads to problems [157] which force administrators to disable it. We run a simple

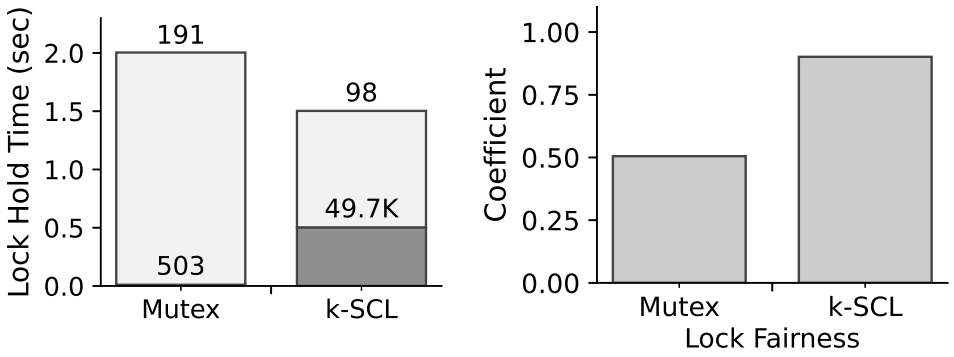
program[2] that repeatedly performs cross-directory renames. We create three directories, where one directory contains one million empty files, and the other directories are empty; each file name is 36 characters. A bully process executes the rename program with *dst* set to the large directory (potentially holding the rename lock for a long time). In contrast, a victim process executes the same program with two empty directories as arguments (thus only needing the lock for a short while).

We use the *ftrace* kernel utility to record cross-directory rename latency, which is plotted in Figure 4.13 as a CDF. The dimmed and the bold lines represent the victim and bully, respectively. The dotted lines show the behavior with the default Linux locks. The bully has an expected high rename latency of 10 ms. About 60% of victim's rename calls are performed when the bully is not holding the global lock and thus have a latency of less than 10 µs. However, about 40% of the victim's calls have a latency similar to that of the bully due to lock contention. If more bullies are added to the system, the victim has even less chance to acquire the lock and can be starved (not shown).

The root cause of this problem is the lack of lock opportunity fairness. To fix this problem, we modified the global rename lock to use the k-SCL implementation. The solid lines in Figure 4.13 show the new results. With k-SCL, almost all of the victim's rename calls have less than 10 µs latency and even its worst-case latency is lower, at roughly 10 ms. Both results arise because the bully is banned for about 10 ms after it releases the lock, giving the victim enough lock opportunity to make progress. If more bullies are added, the effect on the victim remains minimal. We believe this behavior is desired because all tenants on a shared system should have an equal opportunity to utilize the lock.

Figure 4.14 shows the breakdown of the bully and the victim process's lock behavior. We can see that for the mutex, the bully process dominates

[2]while True: touch(src/file); rename(src/file, dst/file); unlink(dst/file);



(a) LHT and Throughput (b) Lock usage fairness

Figure 4.14: **Rename lock performance comparison.** *The figure presents a comparison of two locks – mutex and k-SCL for 2 threads on two CPUs; each has the same thread priority, and one thread is performing rename operations on a directory that is empty while another thread is performing rename on a directory having a million empty files.*

lock usage, and the fairness coefficient is very low. k-SCL penalizes the bully process by providing enough opportunity to the victim process to acquire the lock. Therefore, the victim thread can easily perform around 49.7K rename operations compared to 503 with the mutex version. As the critical section size of the victim program is very small, the non-critical section involving touch and unlink operation dominates the lock opportunity time presented to the victim. Thus, the victim's lock hold time is not quite equal to that of the bully process.

We conduct another experiment using the same rename program to study our approach to inactive thread detection. The bully program runs for the whole duration, while two victims start at 10sec and 20sec, and both exit at 30sec. Figure 4.15 shows the rename latency of the bully during the experiment. After the first victim joins and participates in the lock acquisition process, the bully's latency doubles immediately as there are two threads participating, increasing the penalty time. When the second victim joins, the bully's latency increases further.

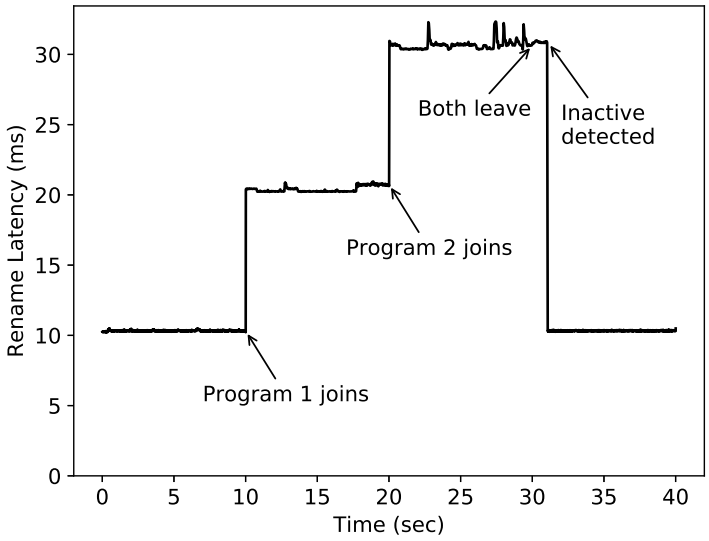


Figure 4.15: **k-SCL inactive thread detection.** *Timeline showing inactive threads detection and how the latency of the bully program varies depending on the number of active threads.*

Once the two victims leave, after 1 second, k-SCL detects that two threads have not participated in the lock acquisition process, initiating the cleanup of those two threads. Once the cleanup completes, as only the bully is actively participating in the lock acquisition process, the latency drops back. Thus, k-SCL can continuously check for active threads, thereby improving the performance of the active threads and increasing the lock utilization. Without this continuous check, the bully would continue to observe a higher latency even though the two victims are not participating.

4.4 Limitations and Applicability

In this section, we discuss the limitations and applicability of SCLs. We start by discussing limitations and their impact on performance. The current design of SCL is non-work-conserving relative to locks. That is, within a lock slice, whenever the lock slice owner is executing non-critical section code, the lock will be unused, even if other threads are waiting to acquire the lock. Since the lock remains unused, performance could be impacted. Threads that have larger non-critical section sizes compared to critical section sizes are more likely to see this effect. One way to alleviate the performance problem is to reduce the size of the lock slice, but at the cost of throughput as shown earlier.

To design a work-conserving lock, one can assign multiple threads to the same class such that multiple threads can acquire the lock within a single lock slice. While one thread is executing the non-critical section, another thread can enter the critical section thereby ensuring high performance. However, it is hard to classify multiple threads statically to one class and hence the lock should have the capability to dynamically classify the threads and create classes. Exploring dynamic classification remains an interesting avenue for future work.

To support a variety of scheduling goals, the implementation of SCL needs to be accordingly adjusted. For example, our current implementation of SCL does not support priority scheduling. To support priority scheduling, the lock should also contain queues and grant lock access depending on the priority.

Scheduler subversion happens when the time spent in critical sections is high and locks are held for varying amounts of time by different threads. As these two conditions can lead to lock usage imbalance, the likelihood of the scheduler subversion problem increases. If there is not much lock contention or all threads hold locks for similar amounts of time, then it might be better to use other simpler locks that have less

overhead. We believe that shared infrastructure represents a competitive environment where multiple applications can be hosted having varied locking requirements. SCLs will play a vital role in such an environment where one application or user can unintentionally or maliciously control the CPU allocation via lock usage imbalance.

SCLs can be used to build *fair-access concurrent data structures*. A fair-access concurrent data structure guarantees fair or proportional access to all the users. We believe that by tracking the lock usage that protects the data structures, one can gauge the data structure usage too. Existing locks can only guarantee correctness in the presence of concurrent operations. However, as they cannot track the lock usage, they are not capable to guarantee fair or proportional access to all the users. On the other hand, as SCLs can track the lock usage, they can guarantee proportional lock usage. Therefore, the data structures built by using SCLs can guarantee fair or proportional access making them fair-access concurrent data structures. These fair-access concurrent data structures can then be used to design kernel and user-space applications.

Even though the focus so far has been on shared infrastructure only, SCLs can be used to design server-like applications that are shared by multiple clients. For example, consider a single instance of MongoDB hosting data belonging to multiple clients having varied requirements. Due to the varied requirement, one or more clients may end up dominating a lock within MongoDB depending on the type or size of the operations. The ability to track lock usage enables SCLs to address the varied requirements while making sure that no single client can dominate the lock usage leading to performance issues.

4.5 Summary & Conclusion

In this chapter, we introduced the concept of lock usage fairness and lock opportunity. Using lock opportunity as a metric to measure lock usage fairness, we studied how traditional locks are unfair in terms of fair usage, as they mostly prioritized lock utilization and (perhaps) delivered lock acquisition fairness. To remedy the problem of scheduler subversion, we introduced Scheduler-Cooperative Locks (SCLs) that track lock usage and can align with the scheduler to achieve system-wide goals. We presented three different types of SCLs that showcase their versatility, working in both user-level and kernel environments.

Our evaluation showed that SCLs ensure lock usage fairness even with extreme lock usage patterns and scale well. We also showed that SCLs can solve the real-world problem of scheduler subversion imposed by locks within applications and the Linux kernel. We believe that any type of schedulable entity (e.g., threads, processes, and containers) can be supported by SCLs and look forward to testing this hypothesis in the future. The source code for SCL is open-sourced and can be accessed at https://research.cs.wisc.edu/adsl/Software/.

So far, locks have been viewed as a cooperative entity; the threads are all part of the same program, and the developer can orchestrate lock usage as they see fit. However, with the rapid adoption of shared infrastructure and varied requirements across users, locks should be viewed as a competitive resource where the locks are accessed by many diverse applications without any isolation or fairness guarantees.

5

Taming Adversarial Synchronization Attacks using Trāṭr

In Chapter 3, we discussed the adversarial aspects of synchronization and introduced two types of attacks – synchronization and framing attacks. By carefully accessing the shared data structures in concurrent infrastructures such as an operating system, hypervisor, or a server, an attacker can launch these two attacks leading to denial-of-services. Synchronization attacks make the victims stall longer, while framing attacks make the victims spend more time in the critical section also.

We also demonstrated several kernel data structures accessed by common system calls – the inode cache and directory cache used by file systems and the futex hash table used for synchronization – are vulnerable to synchronization and framing attacks and demonstrate how an unprivileged attacker can cause throughput reduction of 65-92% to real-world applications in a container-based environment. While the inode cache and directory cache attacks are active attacks i.e. synchronization attacks; the futex table attack is a passive attack, i.e., a framing attack.

Even though all these attacks have the same goal of targeting a synchronization mechanism in the kernel, the way the attacks are launched is entirely different. For the inode cache attack, the attacker breaks the hash function and then runs a simple dictionary attack to expand a target hash bucket by creating thousands of entries such that all the entries

end up in the targeted hash bucket. As the hash bucket expands, the time to traverse the hash bucket increases leading to longer synchronization stalls.

For the futex table, unlike the inode cache attack, the attacker does not target the hash bucket. Instead, the attacker probes all the hash buckets in the futex table to identify a target hash bucket. On identifying the target hash bucket, the attacker creates thousands of threads and parks them in the targeted hash bucket. Thus, any victim that needs to traverse the hash bucket will have to spend more time traversing thousands of entries leading to longer critical section sizes. One should note here that the futex table attack is a framing attack where the attacker turns passive immediately after parking thousands of threads.

The directory cache attack does not attack the mutual exclusion locks but instead attacks the RCU mechanism. An attacker can launch an attack by breaking the dcache hash function or randomly creating millions of entries overwhelming the hash table. In doing so, the RCU read critical section sizes will increase, leading to a more extended grace period. Any thread or process waiting for the grace period to be over will have to wait longer, leading to poor performance.

In this chapter, based on our experience with these attacks, we develop Trātṛ, a Linux extension to defend against synchronization and framing attacks. As the problem is distributed across many kernel data structures, Trātṛ provides a general framework for addressing these attacks using four stages:

- *Tracking:* Trātṛ tracks the contribution to data structure size by each tenant and attaches the user-id information in each object to identify who allocated the objects. This information helps in detecting an attack and perform recovery.

- *Detection:* Trātṛ periodically monitors the synchronization stalls to

detect whether they are longer than expected. If so, using the tracking information, Trātṛ identifies the attacker and initiates steps to mitigate the attack.

- *Prevention:* On detecting an attack, Trātṛ immediately prevents the attack from worsening by blocking attackers from extending target data structures with more elements.
- *Recovery:* With the help of tracking information, Trātṛ takes data structure specific actions to recover baseline performance by isolating or removing the attacker's elements in the shared structure.

We demonstrate the effectiveness of Trātṛ on three such attacks, one each on the file system inode cache, the futex table, and the dentry cache. We show Trātṛ's ability to detect when an attack occurs, prevent it from worsening, and recover performance to baseline (no attack) levels. At steady state, Trātṛ causes only a 0-4% tracking overhead for a variety of applications, and in the absence of an attack, the other stages have less than 1.5% impact on the performance. We also show how Trātṛ can detect and mitigate multiple attacks simultaneously without impacting the performance of the victims.

The rest of this chapter is organized as follows. In Section 5.1, we first discuss how existing solutions are unable to address the adversarial synchronization problem. Then we discuss the design and implementation of Trātṛ in Section 5.2 and evaluate Trātṛ in Section 5.3. We present the limitations of Trātṛ in Section 5.4 and summarize in Section 5.5.

5.1 Mitigating Adversarial Synchronization

Before we start discussing how one can address the problem of adversarial synchronization, let us first recall the synchronization and framing attacks. To launch a synchronization attack, two conditions are necessary:

- Condition S1: A shared kernel data structure is protected by a synchronization primitives such as mutual exclusion locks or RCU that may block.
- Condition S2: Unprivileged code can control the duration of the critical section by either
  - S2_*input*: providing inputs that cause more work to happen within the critical section
    OR
  - S2_*weak*: accessing a shared kernel data structure with weak complexity guarantees e.g., linear).
    AND
  - S2_*expand*: expanding or accessing the shared kernel data structure to trigger the worst-case performance.

In this chapter, we will not focus on the input parameter attacks (S2_*input*). We will deal with attacks that can be launched on data structures having weak complexity guarantees only.

As a framing attack is an extension and refinement on a synchronization attack, the following conditions are necessary to launch a framing attack:

- Condition S1 + S2_*weak* + S2_*expand*: An attacker manages to expand a shared kernel data structure with weak complexity guarantees, i.e., a synchronization attack is in progress or was launched earlier.
- Condition F1: Victim tenants access the affected portion of the shared data structure with worst-case behavior.

By launching a synchronization attack, an attacker actively participates in the attack, and makes the victims stall longer to acquire the lock.

While with the framing attack, an attacker need not participate in the attack but still manages to make the victims spend more time in the critical section while still making other victims stall longer to acquire the lock.

5.1.1 Existing Solutions

Attacks on synchronization primitives can be addressed by interrupting one of the criteria necessary for an attack by the following:

- Breaking condition S1 by using wait-free or partitioned data structures.
- Breaking condition S2_*weak* and S2_*expand* by using avoidance techniques such as universal hashing, balanced trees, or randomized data structures [46].

One may think that by using wait-free data structures, the problem of adversarial synchronization can be solved. Often atomic operations like compare-and-swap (CAS), test-and-set (TAS), fetch-and-increment (FAI), and swap (SWAP) are used to design wait-free data structures [138]. However, for multi-sockets platforms, the performance of these atomic operations is dictated by the cache-coherence latencies leading to poor performance [50]. Moreover, no formal study has been done to understand how wait-free data structures ensure fairness; hence, predicting their behavior in a competitive environment will be hard.

Partitioning is another traditional approach used to design data structures to ensure isolation [122]. However, it is not easy to partition all the data structures. Consider the example of a cache-like data structure that is being shared across all the users. One easy way to partitioning the data is based on who owns or creates the data. Other users who need to access the data can do so by looking at the partition of the user who owns or created the data. However, an attacker can expand his data structure

first and then force another user to load the targeted hash bucket entries. Therefore, not all types of data structures can be partitioned.

Using balanced trees such as red-black trees, AVL trees, and treaps that do not have weak complexity guarantees ($S2\_weak$) is a viable option. However, rewriting the kernel to use balanced trees is tedious [43, 125] and slower than randomized data structures in common cases. Lastly, randomized data structures are also vulnerable to algorithmic complexity attacks [23].

It is not easy to convince developers to use secure hash functions such as SipHash due to performance concerns [44]. We observe around 5-6% performance reduction when we replace the existing hash function in the inode cache with SipHash while running a single-threaded, simple file create workload confirming developers' concerns. Moreover, as we have shown in Section 3.2.3, instead of trying to break a hash function, an attacker can employ other methods like probing each hash bucket to launch an attack. Therefore, relying on strong hash functions alone is not enough to avoid an attack.

To handle framing attacks and prevent victims from accessing expanded data structure (condition F1), rehashing all the entries into a new hash table is possible. However, doing so is invasive to the code and may cause long delays during rehashing leading to varying performance.

5.1.2 Scheduler-Cooperative Locks

Scheduler-Cooperative Locks (SCLs) can mitigate the attacks as they can guarantee lock usage fairness to all the participating users. During a synchronization attack, the attacker aims to dominate the lock usage and prevent victims from acquiring the lock. Thus, SCLs can be useful to break the lock usage domination by the attacker and prevent synchronization attacks.

SCLs are flexible to support any type of schedulable entity such as threads, processes, containers. For these entities, SCLs can track the lock usage and hence can guarantee fair lock usage. To understand the behavior of SCLs when subject to a synchronization attack, we first replace the existing global spinlock `inode_hash_lock` in the Linux kernel's inode cache with k-SCL. Then we run the *IC benchmark* described in Section 5.3 as a victim and launch an inode cache attack from a separate container. Finally, we use the same experimental setup as described in Section 5.3.

With k-SCL, we observe that an attacker cannot identify the superblock address needed to target a random hash bucket in the inode cache during the prepare phase. During the prepare phase, the attacker relies on measuring latencies to identify the superblock address. However, to guarantee fair lock usage, as k-SCL penalizes the attacker, the attacker cannot measure latencies accurately, leading to a failure in identifying the superblock address and not being able to launch an attack.

To study how k-SCL behaves when the attacker can identify the superblock address using other methods, we run the IC benchmark without the prepare phase and directly launch the attack. Figure 5.1 shows the timeline of the throughput of the IC benchmark for the duration of the attack. For comparison purposes, we show the timeline for the Vanilla kernel, i.e., the standard Linux kernel 5.6.42 with and without attack, along with the performance of the k-SCL based kernel.

We observe that k-SCL performs better than the Vanilla kernel when under attack. As the k-SCL rate limits the attacker to guarantee lock usage fairness, the victim's performance improves. However, the performance is around 60% of the Vanilla kernel without an attack. Although k-SCL penalizes the attacker, it cannot break the condition S2_*expand*, and the hash bucket continues to grow at a slower pace. As the hash bucket grows, the victim must wait longer to acquire the lock leading to performance degradation.

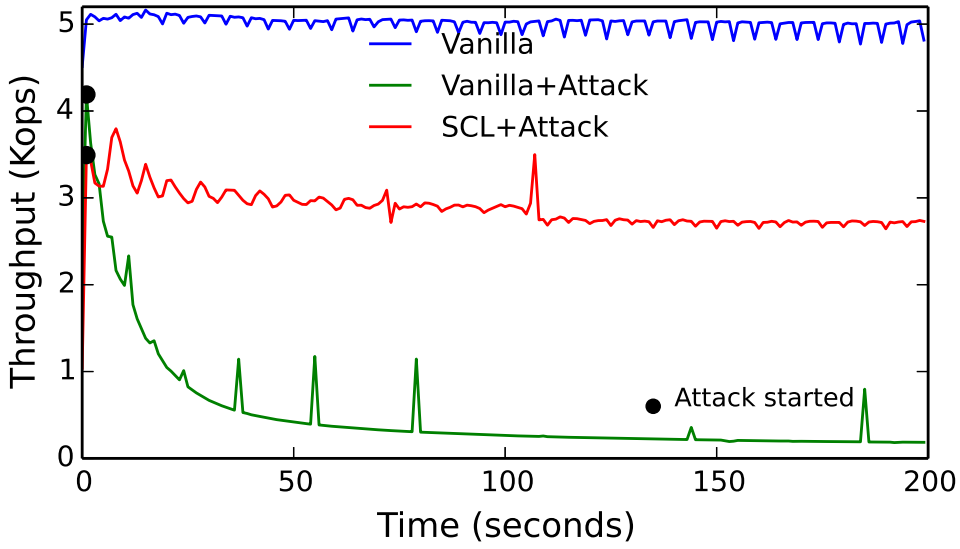


Figure 5.1: **IC benchmark performance comparison for spinlock and k-SCL in Linux kernel.** *Timeline of the throughput showing the impact due to the attack for the Vanilla kernel having spinlock and kernel having k-SCL under the inode cache attack.*

k-SCL will be more effective and ensure better performance when there are more victim threads than attacker threads. As k-SCL penalizes the dominant threads depending on the number of participating threads, the penalty will be higher when there are more victim threads. However, when no victim is present and participating in the lock acquisition process, as k-SCL will remove all inactive threads from the penalty calculation, the attacker will be able to expand the hash bucket without any trouble. Thus, SCLs may not be effective in tackling synchronization attacks.

SCLs will fail to handle the framing attack, too, as they are not aware of the cause of the longer lock hold times. In such a situation, as the attacker is not actively participating, SCLs may treat the victims like the one dominating the lock usage and penalize the victims instead of the

attacker.

As one may have noticed, addressing the framing attacks requires additional steps to repair the shared data structures even after the attacker stops executing to ensure that condition F1 is not met. Merely preventing the continuation of an attack does not stop victims from accessing the affected portion of the data structure.

5.1.3 Summary

While we demonstrated these problems on three data structures, the problem may be widespread as there are hundreds of kernel data structures that may meet the conditions for the attack. We analyze 5429 critical sections protected by 617 locks, a small subsection of the total critical sections in the Linux kernel. We find that 1039 contain loops (19%) and 112 instances (2%) that call `synchronize_rcu()` in the critical section respectively that an attacker can exploit. A more detailed study is needed to understand which characteristics of the critical section can be exploited by an attacker.

Therefore, we take a multi-pronged approach to addressing these attacks. We seek (1) lightweight mechanisms to detect an in-progress attack, followed by (2) a combination of prevention strategies for active attacks to block a malicious actor from continuing an attack, and (3) recovery strategies that seek to restore the data structure to its normal access cost.

5.2 Trāṭr

We now introduce Trāṭr– an extension to the Linux kernel that provides a framework to detect and mitigate synchronization and framing attacks. First, we present the goals for our design and then an overview of Trāṭr design followed by implementing Trāṭr with two recovery solutions.

5.2.1 Goals

We have four high-level goals to guide our design:

- **Automatic response and recovery.** We seek an automated solution to synchronization and framing attacks to reduce administrator effort. While preventing an attacker from further activity may be sufficient for synchronization attacks, framing attacks require a recovery mechanism to restore data structure performance properties.

- **Low false positives and negatives.** As there is a thin line between heavy resource usage and denial-of-service, it can be difficult to determine when an attack occurs. Furthermore, prevention and recovery mechanisms hurt the performance of the attacker by design. Therefore, we seek detection mechanisms relying on multiple signals to avoid false positives and negatives.

- **Easy/flexible to support multiple data structures.** Data structures may require specialized recovery solutions, so a single generic solution is not possible. Hence, it should be easy for the developers to incrementally add protection to targeted data structures as attacks are identified.

- **Minimal changes to kernel design and data structures.** Much effort has been put into selecting and designing kernel data structures such as linked lists, hash tables, and radix trees [116]. Therefore, we want to avoid extensive changes to the kernel design or modifications to hash functions that could lead to performance issues.

5.2.2 Overview

The first step in using Trātṛ is to identify vulnerable data structures used in attacks. We performed this task manually, but it could also be deter-

mined using static analysis or as part of an attack postmortem. After finding such a data structure, our general approach is to track resource usage in a steady-state and detect anomalous resource usage as a sign of attack. On detecting an attack, Trātṛ acts to prevent the attack from continuing and recover from the attack's effects.

For detection, we observe two *detection conditions* common to all three attacks that indicate an attack is in progress:

- *Condition LCS: Long critical section.* Expansion of the data structure causes more work for both victims and the attacker, making the critical section longer.
- *Condition HSUA: High single-user allocations.* A single user has created many entries associated with the data structure.

Neither condition on its own is sufficient to indicate an attack as there may be other reasons for abnormal high allocations or long critical section times, such as interrupt handling. In combination, though, these two conditions can precisely identify attacks. Hence, Trātṛ first checks *critical section size* and if it is too large, then check if one of the users has a majority of the *object allocations* to detect an attack. We choose this order because we believe that checking for long critical sections is easier and less intrusive to other users' workloads than checking for high allocations and traversing the data structure.

Once an attack is detected, Trātṛ responds quickly by preventing the attack from worsening. The system stops the attacker from extending the data structure for a period. Even if the attacker is stopped from allocating more objects, the attacker can access the expanded data structure and continue with the synchronization attack, or the victim can access the expanded data structure in a framing attack.

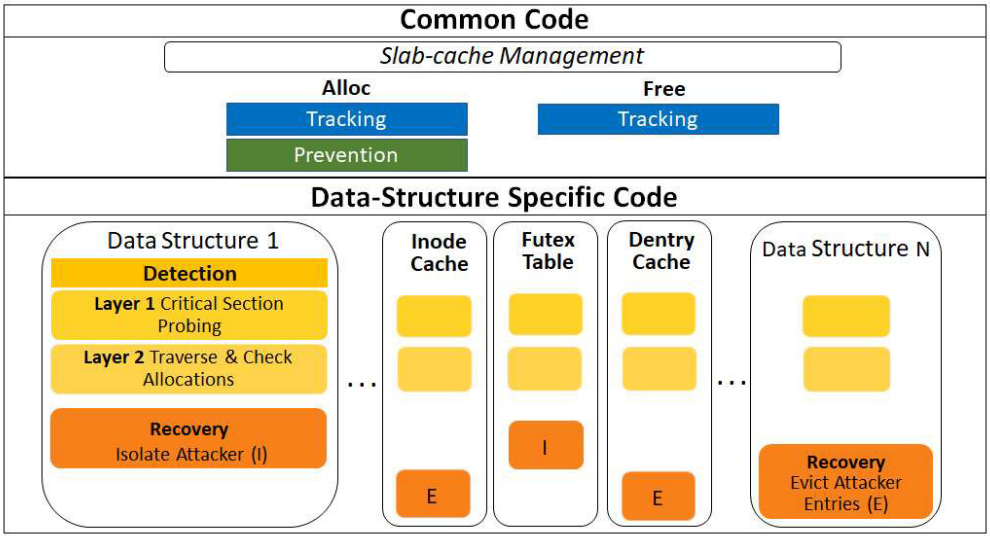


Figure 5.2: **High-level design of Trātṛ.** *Design showing the four mechanisms of Trātṛ. The Tracking and Prevention mechanisms are part of slab-cache management. Layer 1 Detection measures the synchronization stalls to indirectly measure long critical sections. On finding longer stalls, Trātṛ triggers layer two checks if a user has a majority of object allocations. On finding one, Trātṛ identifies that user as an attacker and initiates prevention and recovery mechanisms. The prevention mechanism prevents the attacker from allocating more entries. Depending on the type of data structure, an appropriate recovery is initiated. The upper box shows the common code, while the lower box shows data structure-specific code.*

The final step is recovery, where Trātṛ repairs the data structure to restore its original performance. Here, Trātṛ relies on the type and purpose of the data structure to find an appropriate recovery mechanism.

5.2.3 Design & Implementation

A high-level design of Trātṛ is shown in Figure 5.2. We discuss the design for two data structures — the inode cache and futex table initially and later discuss the directory cache to explain the steps needed to add a new

data structure to Trātṛ. We implement Trātṛ in Linux kernel version 5.4.62.

5.2.3.1 Tracking

The main purpose of having the tracking mechanism is to support the detection and recovery mechanisms. Trātṛ records the allocation and freeing of objects associated with a vulnerable data structure. While allocating an object, Trātṛ (i) tracks the total number of objects currently allocated by the current user, and (ii) stores the user ID in every allocated object. The total number of objects allocated by the current user helps to identify the HSUA condition.

The user ID information is used by recovery mechanism to identify which objects are allocated by the attacker. To attach the information of which user created the object, Trātṛ increases the size of the object by 4 bytes during the slab cache initialization and then stores the current process's user ID at the end of each allocated object.

Linux associates a *slab-cache* with a data structure to manage object allocation and freeing [66, 129]. The Linux slab allocator maintains all the slab-caches and allows the kernel to efficiently manage the objects. In the Linux kernel, the SLUB allocator is the default slab allocator. A slab-cache is a collection of continuous pages into *slabs*. As slabs are used for object allocation and freeing, the kernel does not need to explicitly allocate and release memory for every object allocation.

Instead of writing a separate tracking mechanism for each object, we use the Linux kernel slab-cache infrastructure to track the objects. We modify the common slab-cache management code to selectively record the relevant information for slab caches associated with the vulnerable data structures. The total objects allocated per user for each slab-cache is stored in a global hash table, updated during allocation and free operations. For our prototype, Trātṛ tracks four objects associated with the inode cache, the futex table, and the directory cache and summarized in

Data Structure	Tracking		Detection	Recovery
	Slab-cache	Object	Primitives probed	
Inode cache	ext4_inode_cachep fuse_inode_cachep	ext4_inode_info fuse_inode	inode_hash_lock	Evict
Futex table	task_struct_cachep	task_struct	futex_hash_bucket.lock	Isolate
Dentry cache	dentry_cache	dentry	RCU	Evict

Table 5.1: **Implementation summary of Trāṭṛ.** *Implementation details of the four slab-caches and three data structures that Trāṭṛ defends against the synchronization and framing attacks.*

Table 5.1.

Tracking is part of the critical path for object allocation and freeing, adding a few more instructions. However, as we show later, this overhead is minimal. Also, Trāṭṛ increases memory consumption through space in each object for the user ID and the array to track per-user allocations.

One should note here that accounting can be performed across a wide range of entities. For example, accounting can be performed at a fine-grained level such as per-thread, per-process, or a coarse-grained level, such as per-user or per-container level. For our work, we choose per-user accounting to tag all the processes and threads as an attacker and avoid situations where an attacker can deploy a multi-threaded attack. However, with simple code changes, any level of accounting is possible.

5.2.3.2 Detection

The primary objective of the detection mechanism is to check whether detection conditions (LCS & HSUA) meet. When they meet, Trāṭṛ flags an attack and initiates prevention and recovery. For effective detection, Trāṭṛ adopts a two-layered approach. The first layer checks for the LCS condition, while the second layer checks for the HSUA condition and identifies the attacker. A separate kernel thread performs the layered checks and performs recovery (discussed later) for each slab cache. Figure 5.3 shows

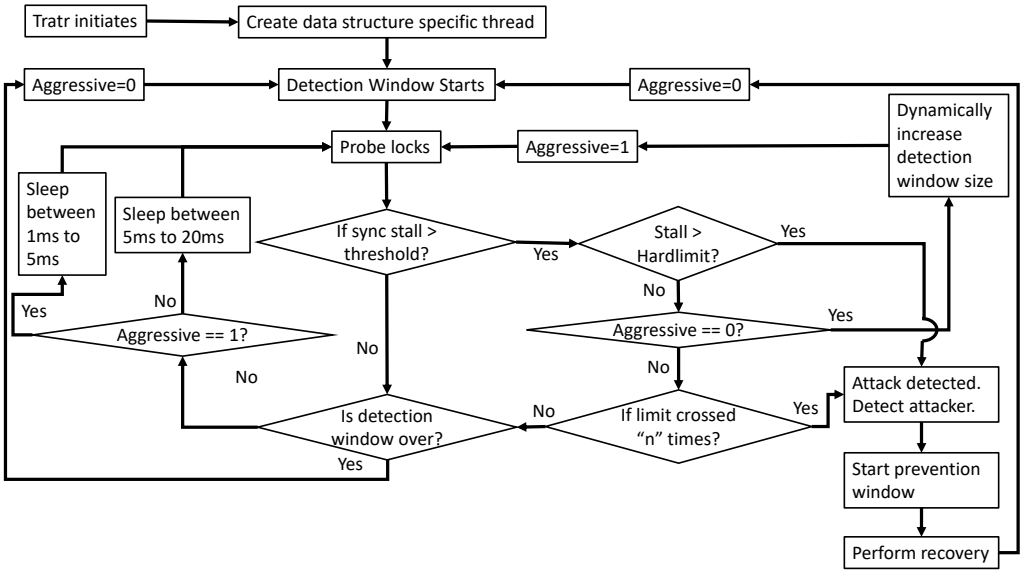


Figure 5.3: **Flowchart of the kernel thread associated with a data structure.** *The kernel thread that is associated with a data structure performs the detection and recovery mechanism. As part of the detection mechanism, the thread probes the synchronization primitives. If a synchronization stall is more than the threshold, appropriate action is initiated. Upon detecting an attack, the thread executes the TCA check to identify the attacker and initiate prevention window. Lastly, the thread initiates the recovery of the data structure.*

the flowchart of the kernel thread that performs the detection and recovery for each slab cache.

The first layer, the *Critical Section Probing* (*CSP*) check, detects if the critical section associated with the data structure is long by probing the locks or RCU grace period size. The check periodically tries to acquire the synchronization primitive and measures the synchronization stalls. If under attack, the stalls will be longer than expected, determined by a threshold. Table 5.1 shows the locks that Trātṛ probes to detect the inode cache and futex table attacks.

Trātṛ uses a 100 microseconds threshold to probe the inode cache and futex table locks. We calculate this value by assuming an even distribu-

tion of the objects in the hash table and all CPUs participating in the lock acquisition process. One must note that the threshold limits may differ depending on the critical section sizes for other locks. Neither the inode cache nor the futex table use RCU. From here on, we will use only locks for our discussion for simplicity. However, the same design applies to the RCU mechanism. We note here that as Trātṛ uses threshold limits to determine if a synchronization primitive is under an attack, badly configured applications or stress testing scenarios may be falsely detected as an attack (false positives).

By probing the lock several times, Trātṛ confirms that the lock is held for long-time multiple times and not just once during the probing period. This reduces the chances of wrongly detecting an attack. We call this probing period as *probing window* when the lock is probed multiple times to detect an attack. If Trātṛ finds that the synchronization stalls are more than the threshold period several times, an attack is flagged, which will initiate the second layer check. Thus, the probing window provides a boundary within which Trātṛ decides if a synchronization primitive is under an attack.

The CSP check interacts with the synchronization primitive, thereby interfering with the normal user operations that want to also acquire the lock. While probing the lock, Trātṛ just acquires the lock and then immediately releases it to minimize the impact of the interference. Additionally, instead of continuously probing, Trātṛ probes the synchronization primitive after a time gap. This time gap is between five and twenty milliseconds and is randomly chosen. The kernel thread sleeps between the probes to let the user-application threads run and do not interfere with the applications. The size of the probing window is randomly calculated and varies between 1 second and 5.3 seconds under normal conditions. Figure 5.4 shows how Trātṛ probes the lock within the probing window.

While probing, if Trātṛ detects that the synchronization stall is more

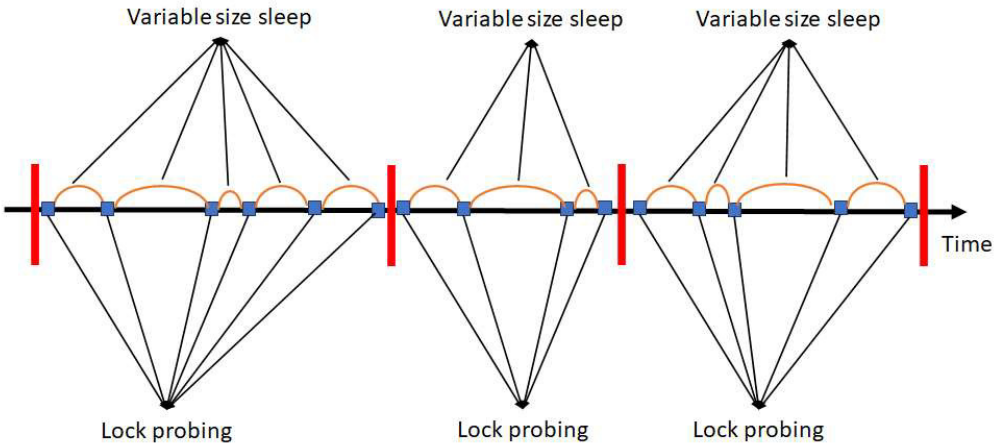


Figure 5.4: **Probing window behavior under normal conditions.** *Under normal conditions without an attack, during the probing window, after probing a lock once, the kernel threads sleep for 5 to 20 milliseconds. Note that the size of the probing window is randomly chosen.*

than the threshold, it dynamically expands the probing window size. The new expanded probing window size is anywhere between 8 to 13.3 seconds. It does so to increase the likelihood of detecting an attack during the same probing window itself. Additionally, Trātṛ also increases the probing frequency to ensure that an attack is detected. So, instead of the regular time gap of five to twenty milliseconds, Trātṛ changes the time gap to anywhere between one millisecond and five milliseconds. Figure 5.5 shows how Trātṛ behaves within a probing window when the synchronization stall is more than the threshold. The idea of aggressively probing is to probe the lock multiple times within a single time slice compared to just probing the lock once within a single time slice when not under an attack.

As the size of the probing window and the time gap between probing is randomized, Trātṛ makes it difficult for a defense-aware attacker to launch an attack. A defense-aware attacker may not know when to launch an attack and when to stop to remain undetected. To remain undetected,

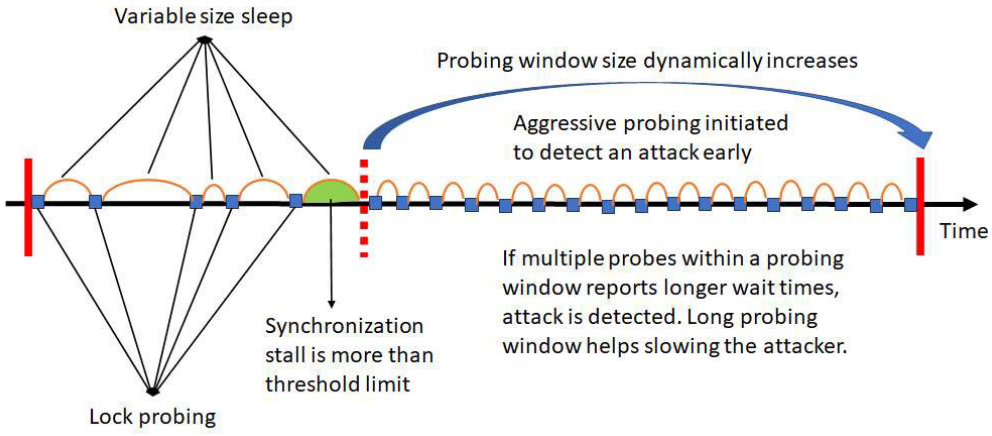


Figure 5.5: **Probing window behavior when under attack.** *When an attack is ongoing, during the probing window, upon identifying that one synchronization stall is more than the threshold limit, Trāṭr dynamically increases the probing window size and aggressively probes the synchronization primitive to detect an attack early.*

the attacker will have to ensure that it cannot hold the lock beyond the threshold and cannot repeat it multiple times within the same probing period. This significantly slows down the attacker and foils the plan to launch an attack.

Even if an attacker very slowly manages to expand the data structure, Trāṭr has a hard limit set for the synchronization stalls, beyond which an attack will be detected immediately. This hard limit check signifies that under no circumstances a given lock will be held for a very long duration. On breaching the hard limit, an attack is flagged.

Trāṭr initiates the second layer, the *Traverse & Check Allocations* (*TCA*) check whenever the first layer flags an attack. In this layer, Trāṭr first traverses the data structure associated with the synchronization primitive and uses the 4-byte user ID embedded in each object to determine each user's total number of allocations. If a particular user has allocated the majority of the entries, Trāṭr flags that user as an attacker and passes the

attacker's identity to the prevention and recovery mechanisms. Flagging the users holding the lock longer is insufficient to determine the attacker, as it may end up tagging the victim of a framing attack. Thus, the embedded user ID information helps Trātṛ in identifying the attacker accurately. For the inode cache and futex table, Trātṛ selects the bucket with the most entries for traversal.

5.2.3.3 Prevention

We identify two approaches to mitigating attacks. First, attackers can be rate limited by stalling them when they try to allocate memory to expand the vulnerable data structure (stopping condition S2_*expand*). Second, the system can terminate or suspend an attacker's container, also stopping condition S2_*expand*. For some data structures, killing the container may trigger clean-up, which stops condition F1 as well. However, we believe killing the container is not appropriate as it may lead to application-level corruption when a user is wrongly identified as an attacker. Suspending the attacker is not appropriate as the attacker may hold locks that are part of the user-space libraries and can impact the victims. Therefore, we opt for the first approach to rate-limit the attacker while allocating memory. We believe that badly configured applications or stress tests [11] may be detected as attackers even though their intention is otherwise.

The prevention mechanism uses the existing slab-cache infrastructure to prevent the attacker from expanding the existing attack or launching future attacks. After identifying the attacker, Trātṛ blocks the attacker from allocating more objects from the slab cache for a specific period, called the *prevention window*. Threads from the attacker are put to sleep until the window expires. This means attackers cannot create new in-memory inodes for the inode cache, blocking them from opening/creating files. For the futex table, this means attackers cannot create new threads. Note that Trātṛ does not fail or stop allocation requests with the

ATOMIC flag, as the flag denotes that the process cannot be blocked. However, we find this flag is rarely used with vulnerable kernel data structures.

Trātṛ maintains a prevention window size for each user separately. Trātṛ initializes the window size to 1 second and then increases the window size depending on how frequently Trātṛ detects the user as an attacker. Trātṛ calculates the growth of the prevention window size by looking at when was the last time the user was marked as an attacker. For an attacker who continuously tries to launch an attack, the growth factor will be high as the attack will be frequent. On the other hand, if a victim is wrongly identified as an attacker a few times, the prevention window size will not grow rapidly as the victim is not aggressively engaging; the victim will resume work without getting stalled for too long.

5.2.3.4 Recovery

With only prevention in place, victims of a framing attack continue to observe poor performance as the expanded data structure still exists. Moreover, attackers can continue to access the expanded data structure and hold the lock. Therefore, recovery is necessary to restore the performance to normal. Trātṛ offers two solutions to design recovery solutions that can support different data structures.

One solution deals with cache-like data structures where the presence or absence of an entry does not impact correctness. For such caches, like the inode cache, Trātṛ evicts all the entries belonging to the attacker. Thus, victims do not lose much performance from the eviction of the attacker's entries, as they typically do not reference those entries. Furthermore, this approach breaks condition F1 as victims no longer traverse the attacker's entries. Implementing eviction for inode cache is straightforward as we reuse existing code. Trātṛ iterates through the file systems in use (currently it supports fuse and ext4) to enumerate all inodes and drop those

allocated by the attacker.

The other solution deals with non-cache data structures where correctness is important. For example, for the futex table, threads must be present on the waitlist to correctly implement synchronization. As each entry in the waitlist is a waiting thread, dropping any entry may leave the threads waiting, leading to problems. Thus, evicting the entries does not work for futex table like data structures.

With these data structures, we observe that in many instances, the data structure is used as a convenient mechanism to manage and group data from all processes, but a single container only accesses that data. For example, for the futex table, even though the waitlist is shared across processes, the processes only access entries that belong to their futex variables.

For such data structures, Trātṛ *partitions the entries* so that victims and attackers use separate, parallel structures, and victims do not have to traverse the attacker's entries. This isolation breaks the condition F1. Trātṛ walks the data structure, identifies entries allocated by the attacker, and moves those entries out of the primary structure to a new *shadow structure*.

On subsequent access, victims only access the original primary structure, while attackers only access the shadow structure. This ensures the number of attacker's entries cannot impact the victims. Trātṛ dissolves the shadow structure once the prevention window ends. We note that partitioning may not work with cache-like data structures, as it could create multiple copies of entries allocated by both victim and attacker and lead to inconsistencies.

Apart from ensuring that the attacker cannot expand the data structure, using a preventing window also helps with the timely recovery without the attacker's interference. The attacker can continue with the synchronization attacks by accessing the expanded data structure even though the attacker cannot allocate more objects. This poses a challenge

to the recovery mechanism, which may need to acquire the synchronization primitive. Thus, Trātṛ prevents the attacker from acquiring the synchronization primitive until the recovery completes. For nested synchronization, Trātṛ needs to prevent the acquisition of the highest-level primitive. If the attack is launched again, Trātṛ will detect the attack and follow the prevention and recovery mechanisms to mitigate the attack.

5.2.3.5 Adding a New Data Structure

Analyzing all existing data structures to check for vulnerabilities is a long process. Therefore, it is important to add more structures to Trātṛ incrementally on finding a new vulnerability. Adding a new data structure is a two-step process.

In the first step, the developer must pass a flag to the memory management system when creating the associated slab-cache at boot time. Then, whenever the slab-cache is initialized, the flag will enable tracking of the objects and start maintaining the accounting information for each user. One should note here that enabling the tracking alone is not enough to detect and mitigate the attacks. However, by enabling the tracking, one can use the accounting information to understand each user's object usage.

The second step is to implement the CSP checks, TCA checks, and the recovery procedure for the added data structure. As mentioned earlier, the threshold limits used during CSP checks will have to be identified by figuring out the size of critical sections that the synchronization primitive protects. Then depending on the number of CPUs available in the system, which is not constant for all the systems, the developer can calculate the maximum synchronization stalls due to contention; and use it to arrive at the threshold limits.

TCA and recovery procedures can be implemented depending on the type of the data structure and is straightforward compared to the CSP

checks. In Section 5.3.5, using the example of dcache, we show the effort needed to add a new data structure.

5.3 Evaluation

In this section, we evaluate the effectiveness, performance characteristics, and responsiveness of Trātṛ. We show that Trātṛ incurs low overhead and can quickly detect and mitigate the attack using microbenchmarks, benchmark suites, and real-world applications. We also illustrate how Trātṛ works in the real world by running multiple containers hosting different applications and launching the earlier discussed attacks.

We perform our experiments on a 2.4 GHz Intel Xeon E5-2630 v3 having two sockets; each socket has eight physical cores with hyper-threading enabled. The machine has 128 GB RAM and one 480 GB SAS SSD. The machine runs Ubuntu 20.04 with Trātṛ built on kernel version 5.4.62. All the applications and benchmarking tools used for the experiments are run as separate Docker containers.

We use two different kernels to evaluate Trātṛ. We label the standard Linux kernel 5.6.42 as *Vanilla*. The kernel having Trātṛ with all the four mechanisms enabled is *Trātṛ*. Additionally, to understand various aspects of Trātṛ, we use multiple configurations within Trātṛ. Trātṛ with just tracking feature enabled is *Trātṛ-T*, and with only tracking, detection, and prevention enabled is *Trātṛ-TDP*. For the overhead experiments, Trātṛ-T tracks all the slab-caches. For any experiment that involves an attack, we add the suffix *+Attack* to note that a kernel is under attack.

5.3.1 Overall Performance

An attacker can employ complexity attacks to turn synchronization primitives adversarial, leading to poor performance. In this section, we use microbenchmarks to show how Trātṛ resists attacks and the impact on

	Throughput (ops)		
	Vanilla	**Vanilla + Attack**	**Trāṭr + Attack**
IC Benchmark	4,982	202	4,884
FT Benchmark	48.95M	0.97M	48.22M

Table 5.2: **Performance of two benchmarks.** *Observed Throughput at the end of the experiment of the IC and FT benchmarks for Vanilla kernel without an attack, with an attack for Vanilla & Trāṭr.*

the victim's performance with or without Trāṭr. We run two microbenchmarks – one each for the inode cache and the futex table, respectively, that we call the victims. We run the experiment for 300 seconds on both Vanilla and Trāṭr kernels to compare the performance with and without an attack. We allot 8 CPUs and 8 GB of memory to the victim and the attacker container.

Recall that the attacker launches an inode cache attack by first identifying the superblock pointer; then target a hash bucket by creating files whose inode number maps to the targeted hash bucket. On the other hand, while launching a futex table attack, the attacker targets the futex table by allocating thousands of futex variables and then probes the hash buckets to identify a busy bucket. Once found, the attacker parks thousands of threads on that hash bucket. As the attack is a framing attack, the attacker turns passive after parking the threads.

The *IC benchmark* associated with the inode cache creates an empty file every 100 microseconds. We measure the throughput (number of inodes created per second), and the latency of each file create operation.

The *FT benchmark* associated with the futex table creates 64 threads that run in a loop, where each thread acquires a lock, increments a shared counter for 100 microseconds, and then releases the lock. We read the counter value every second to calculate the throughout and measure the latency to release the lock using systemtap in a separate experiment.

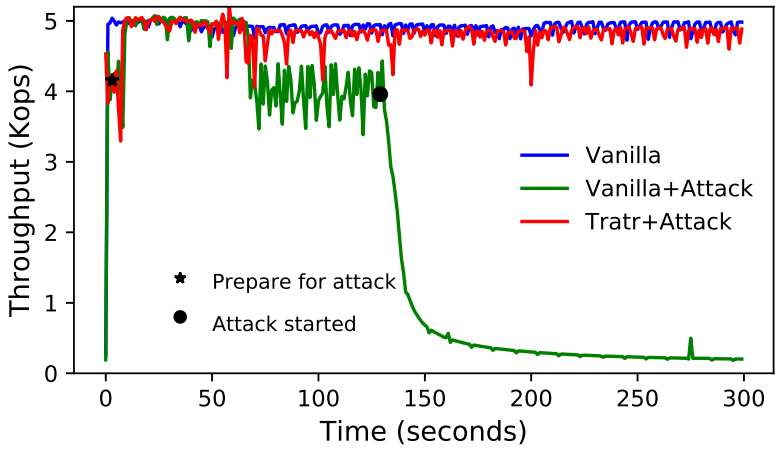
Table 5.2 shows the performance comparison of both the IC and FT benchmark respectively for three experiments run on the Vanilla and Trāṭr kernel. We use the performance of the Vanilla kernel without an attack as the baseline performance. When the Vanilla kernel is under an attack, at the end of the experiment, the throughput of both the IC and FT benchmarks drops by 95.95% and 98.06% respectively, leading to a large denial-of-services. With Trāṭr, there is no significant performance degradation compared to the baseline.

5.3.1.1 Inode Cache Attack

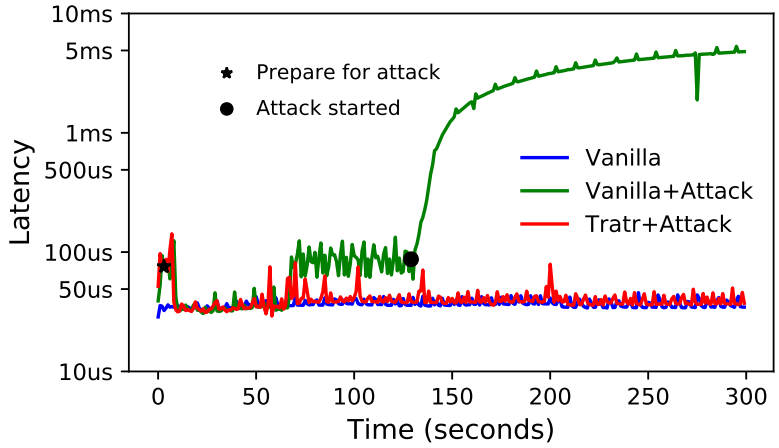
Figure 5.6 shows the throughput and average latency timeline of the IC benchmark. When the attacker is probing the hash buckets to identify the superblock pointer during the prepare phase, the Vanilla kernel performance drops and varies. After identifying the superblock pointer, as the attack starts, the throughput drops significantly and stays the same until the end of the experiment. As the attacker holds the global inode cache lock for a long duration, the IC benchmark must wait longer to acquire the lock. As more and more entries get added to the hash bucket, the wait-time increases, thereby increasing the latency of the operations.

On the other hand, Trāṭr detects the attack while the attacker is still preparing for the attack. As Trāṭr prevents the attacker from allocating more objects, it cannot expand the hash bucket breaking condition $S2\_expand$. As part of the recovery, Trāṭr removes all the entries from the hash bucket. As the attacker relies on the latency measurement to identify the superblock address, due to the prevention and recovery mechanisms, Trāṭr foils the attack while the attacker is still preparing for the attack. As the attack is never started, there is no change in the throughput for the IC benchmark.

During the experiment, as the inode cache attack is ongoing, immediately after the prevention window ends, the attacker again tries to create

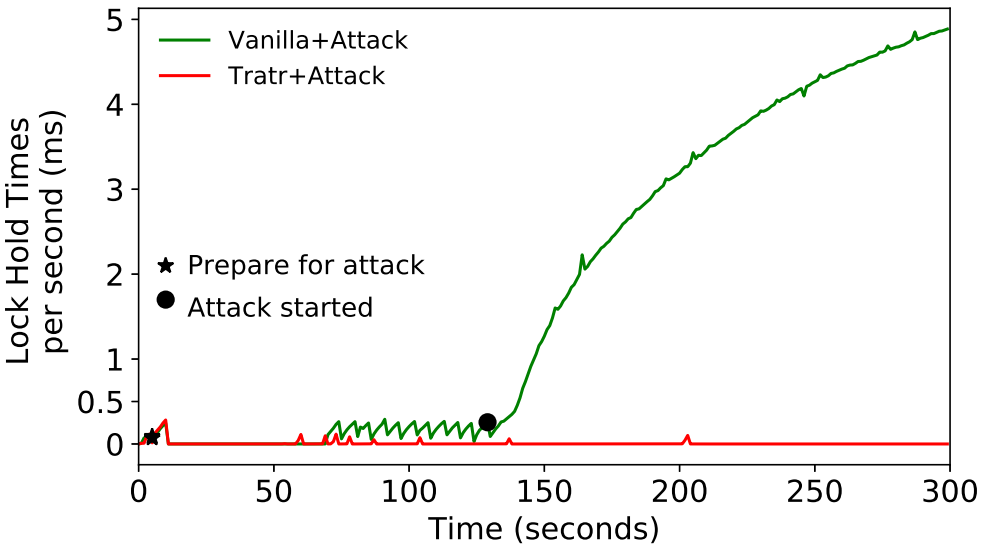


(a) Throughput

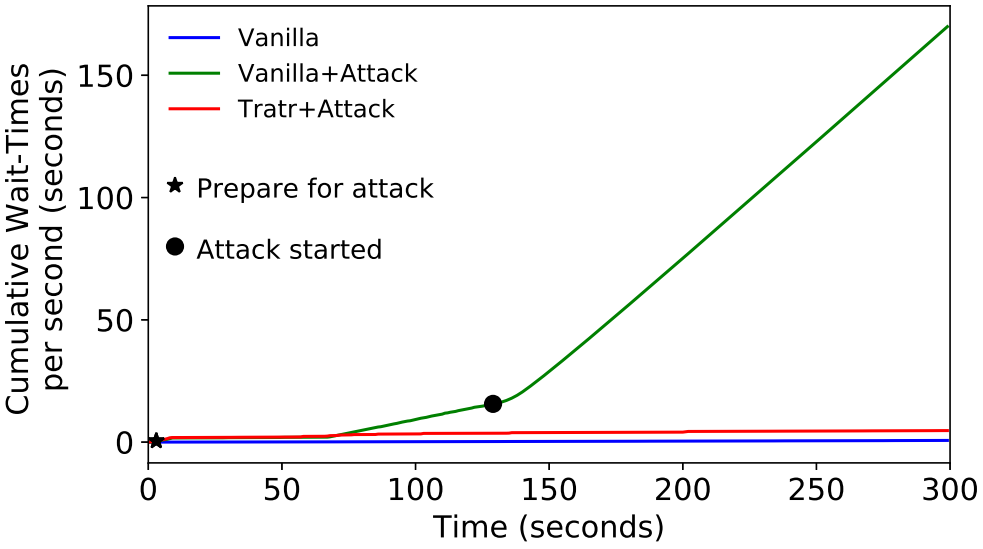


(b) Latency

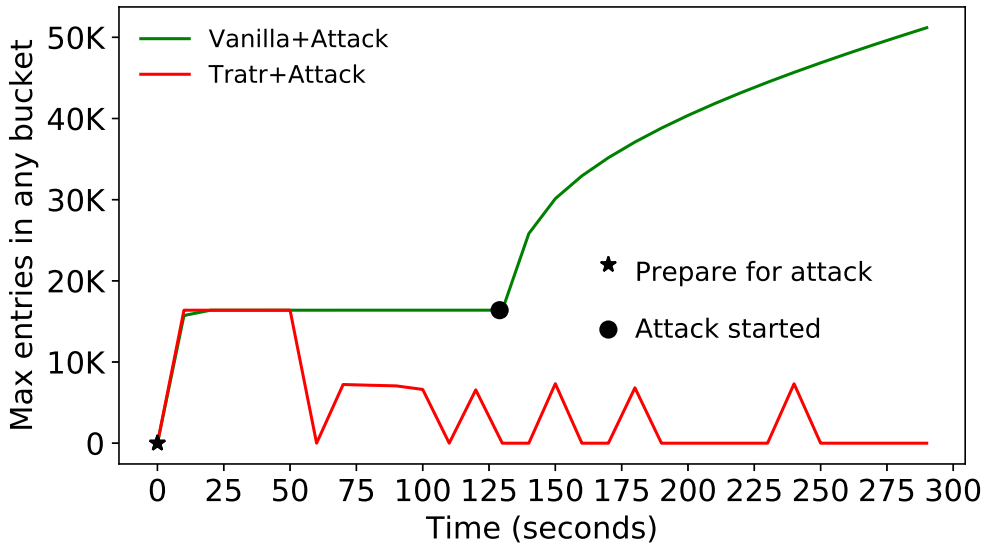
Figure 5.6: **IC benchmark performance without attack, with attack and with Trāṭr.** *(a) Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trāṭr kernel. With Trāṭr, the attacker is not able to launch an attack. (b) Timeline of the average latency observed every second while creating the files.*



(a) Lock hold times of the attacker for Vanilla and Trātṛ kernel.



(b) Cumulative wait times of the victim for Vanilla and Trātṛ kernel.



(c) Maximum number of attacker entries in any bucket for Vanilla and Trātṛ kernel.

Figure 5.7: **Internal state of inode cache when under inode cache attack.** *The graphs present an overall picture of the inode cache when an attacker is launching the attack. In particular, the timeline shows the lock hold times of the attacker, the cumulative wait times to acquire the inode cache lock, and the maximum number of entries of the attacker for the Vanilla kernel and Trātṛ. The victim is running the IC benchmark.*

entries during the prepare phase of the attack. Therefore, there are a few drops in the performance periodically. However, the moment the threshold limit is crossed while probing for the inode cache lock, Trātṛ will flag the attack and immediately initiate preventive and recovery mechanisms. Another point to note is that the attack can turn into a framing attack if the victims access the targeted hash bucket.

Internal state of the inode cache. Figure 5.7 shows the internal state of the inode cache while the attack is going on. Figure 5.7a shows the lock hold times of the attacker for Vanilla and Trātṛ when under attack. We observe that as the attacker continues to expand the targeted hash bucket

for the Vanilla kernel, the lock hold times also continue to increase. We do not show the victim's lock hold times as they are very small compared to the attacker's lock hold times.

On the other hand, Trātṛ can detect and mitigate the attack and therefore manage to keep the attacker's lock hold times under the threshold limits. As a result, we only observe a slight increase in the lock hold times when the attack is started. At this point, the attack is still in the prepare phase. However, as the preventive and recovery measures kick in once the attack is detected, the attacker cannot impact the victim's performance.

Figure 5.7c shows the maximum number of attacker's entries in any bucket in the inode cache. This result corroborates how the preventive and recovery mechanisms never allow the attacker to expand the targeted hash bucket. Whenever the attacker is able to expand the targeted hash bucket, as the synchronization stalls cross the threshold limits, the recovery mechanism evicts all the attacker's entries. On the other hand, for the Vanilla kernel, the number of entries in the targeted hash bucket increases, leading to a denial-of-service attack.

Figure 5.7b shows the victim's cumulative wait times to acquire the inode cache lock. We observe that without an attack, as there is no competition to acquire the inode cache lock, the cumulative wait time is negligible. On the other hand, when under attack, the victim's wait times continue to grow as the attack progresses. As the attacker starts to dominate the lock usage, the victim thread has to wait longer to acquire the lock leading to poor performance.

However, as Trātṛ can quickly detect and mitigate the inode cache attack, there is no increase in the victim's wait times as the attacker cannot dominate the lock usage by targeting a hash bucket. The preventive and recovery mechanisms help in foiling the attacker's plans.

Economic impact. For the Vanilla kernel, without an attack, the victims have to wait for roughly 1% of the total runtime to acquire the inode cache

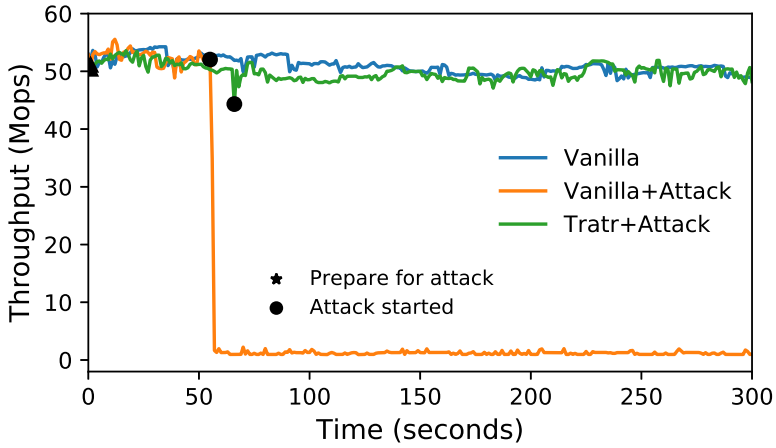
lock. On the other hand, when under attack, as the inode cache attack is a synchronization attack, the victims will observe very high wait times. As the inode cache lock uses spin-waiting, the victims will be spending the CPU cycles for which the Cloud vendors will be charging the victims. We observe that the victim spends 85% of the total runtime waiting to acquire the inode cache lock. So, the victim ends up paying more without doing any useful work. For the victim, a synchronization attack impacts the performance and makes the victim suffer economically.

With Trātṛ, due to early attack detection, the victim's have to wait for around 5% of the total runtime. This increase in the wait time compared to the baseline is because Trātṛ needs to probe the locks periodically and perform recovery and hence has to acquire the lock to clean up the inode cache.

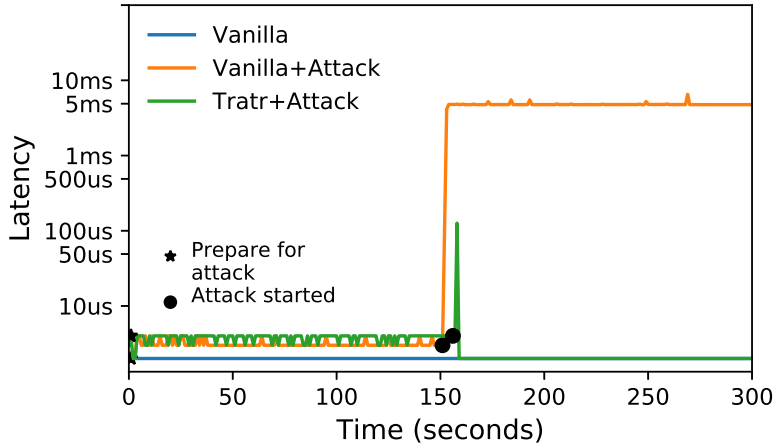
5.3.1.2 Futex Table Attack

Figure 5.8 shows the throughput and average latency timeline for the FT benchmark. As the futex table attack increases the time a thread spends while releasing the lock, we show the latency of releasing the lock in the latency timeline by conducting another similar experiment and using Systemtap to measure the time to release the lock. Once the attack starts, the throughput of the FT benchmark drops significantly and continues to stay the same. At the same time, the latency of releasing the lock increases by 1200 times from 3-4 µs to 4.8 ms.

On the other hand, Trātṛ detects the attack quickly and mitigates the attack by isolating the entries of the attacker breaking condition F1. As the FT benchmark does not access the entries allocated by the attacker, the throughput returns to the baseline level once the recovery happens. Overall, the FT benchmark sees around a 1.5% drop in the throughput. This negligible throughput drop shows the effectiveness of Trātṛ and shows how quickly the attack is detected and mitigated.

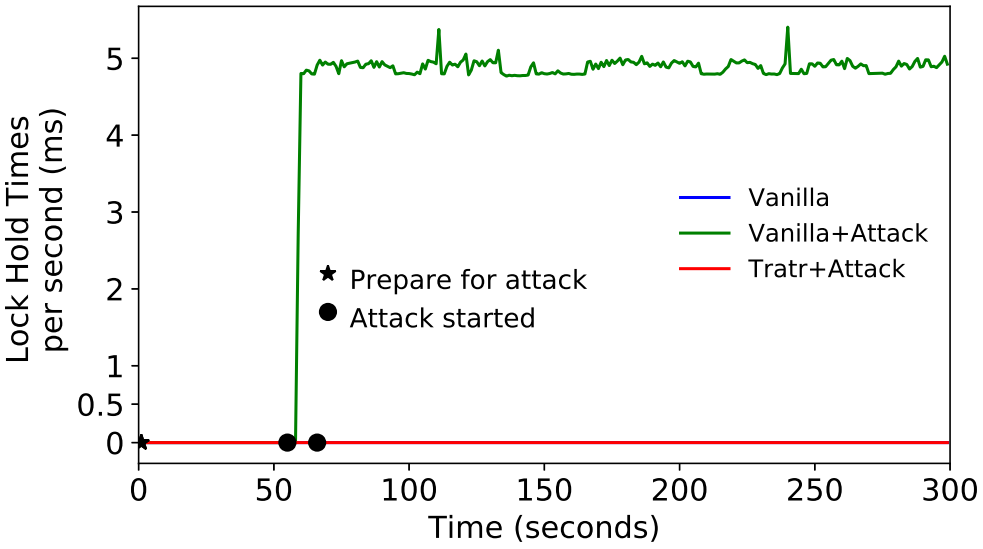


(a) Throughput

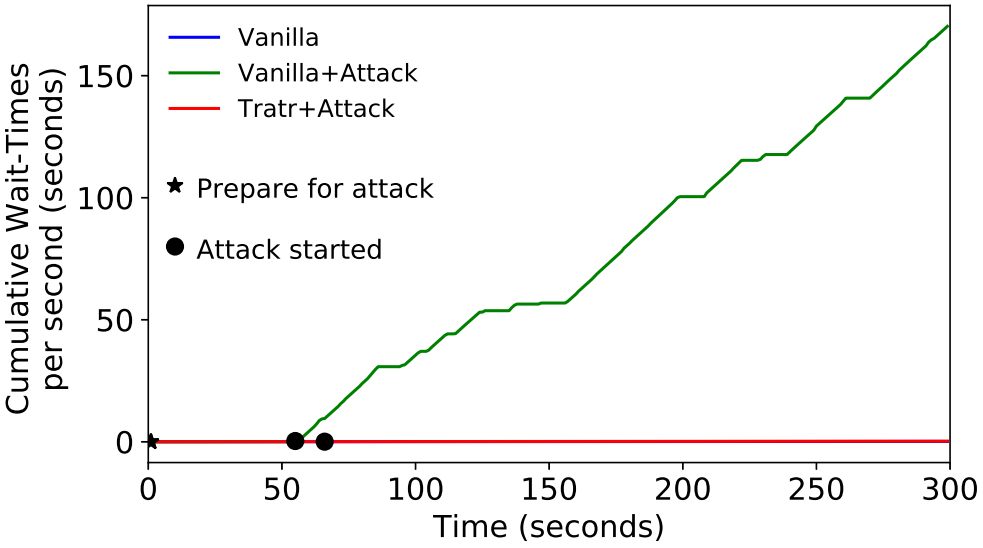


(b) Latency

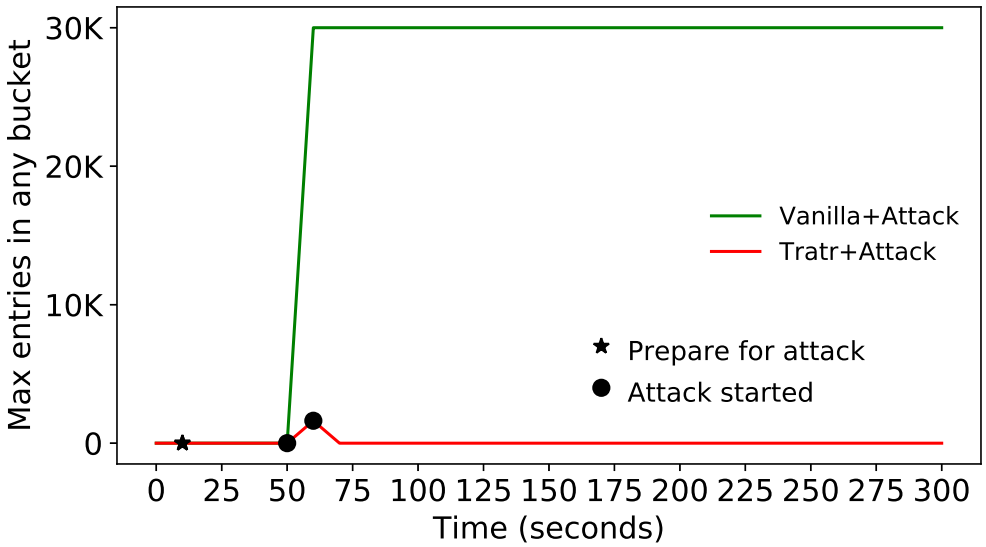
Figure 5.8: **FT benchmark performance without attack, with attack and with Trāṭr.** *(a) Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trāṭr kernel. (b) Timeline of the average latency for another experiment of the time to release the lock.*



(a) Lock hold times of the victim for Vanilla and Trātṛ kernel.



(b) Cumulative wait times of the victim for Vanilla and Trātṛ kernel.



(c) Max number of entries of the attacker in any bucket for Vanilla and Trātṛ kernel.

Figure 5.9: **Internal state of futex table when under futex table attack.** *The graphs present an overall picture of the futex table when an attacker is launching the attack. In particular, the timeline shows the lock hold times of the victim, the cumulative wait times to acquire the hash bucket lock, and the maximum number of entries of the attacker. The victim is running FT benchmark.*

Internal state of the futex table. Figure 5.9 shows the internal state of the futex table while the attack is going on. As the futex table attack is a framing attack, the attacker remains passive after expanding the targeted hash bucket. Figure 5.9a shows the lock hold times of the victim for Vanilla and Trātṛ with and without an attack. For the Vanilla kernel, once the attack is launched, the lock hold times of the victim increase 25000X times from hundreds of nanoseconds to 5 milliseconds.

On the other hand, Trātṛ can detect and mitigate the futex table attack by moving the attacker's entries to a shadow bucket. Furthermore, by isolating the attacker and the victim, Trātṛ manages to keep the lock hold times of the victim similar to the baseline level.

Figure 5.9c shows the maximum number of attacker's entries in any bucket in the futex table. For Trātṛ, once an attack is detected, the recovery mechanism kicks in and moves all the attacker's entries to the shadow bucket. However, for the Vanilla kernel, as part of the attack, the attacker parks thousands of threads on the targeted hash bucket, increasing the total number of entries to 30000. As the attacker turns passive, the number of entries stays the same until the end of the experiment.

Figure 5.7b shows the cumulative wait times to acquire the futex table hash bucket lock. Under the framing attack, the victim's wait times continue to grow as the attack progresses. This is because multiple victim threads wait to acquire the lock longer, which another victim thread is holding on to for a long time. On the other hand, due to the quick detection and recovery, the victim hardly sees any substantial increase in the wait times for Trātṛ. Thus, the victim's wait times for Trātṛ is similar to the baseline level.

Economic impact. Framing attacks not only make the victims wait longer to acquire the lock but also make them spend more time in the critical section also. We observe that when under attack, the victim's CPU usage shoots up by 2.44X times more than the baseline case. As the victim has to traverse the expanded hash bucket every time it accesses the hash bucket, the critical section size increases, leading to more CPU usage. For Trātṛ, the total runtime is around 1.2% more than the baseline level. By detecting the futex table attack quickly, Trātṛ can reduce the increase in the total runtime. Without Trātṛ, the victim will end up paying 2.44 more for the extra CPU usage and also observe about 98.06% drop in the performance.

We also observe that the victim spends around 17.8% of the total runtime waiting while acquiring the lock. On the other hand, for the baseline case, the victim spends less than 0.01% of the total runtime waiting to acquire the lock. Similarly, with Trātṛ too, the total time spent waiting is less than 0.01%. As Trātṛ quickly detects and performs recovery, there is

minimal impact on the total wait time.

5.3.2 Performance of Trāṭr Components

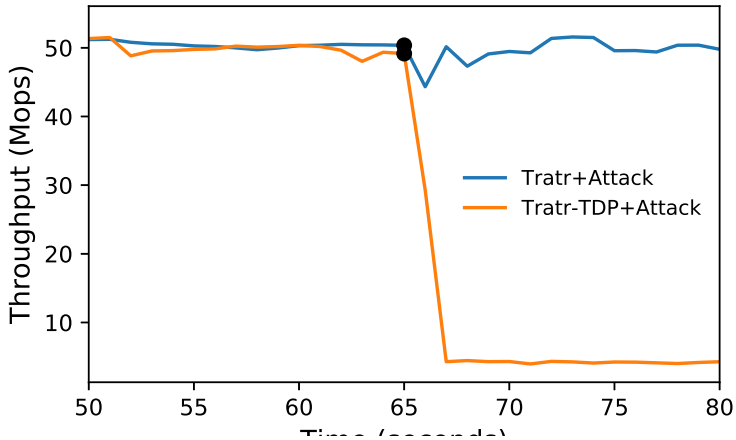
We evaluate the responsiveness of Trāṭr in detecting attacks and how the prevention and recovery mechanisms help to mitigate the attack. We rerun the IC benchmark without the preparation phase. We use the same data from the previous experiment for the FT benchmark but present a zoomed version for better understanding.

5.3.2.1 Detection

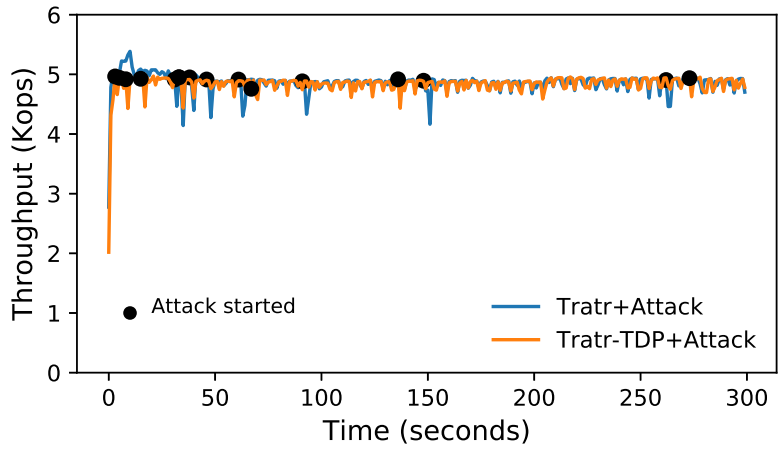
Early detection of an attack is important to reduce the impact on the victim's performance. As Trāṭr probes the synchronization primitives several times during the probing window, it can immediately identify that an attack is ongoing and take punitive actions to prevent the damage caused by the attack. We present the timeline of the throughput of the IC and FT benchmark in Figure 5.10.

For the FT benchmark shown in Figure 5.10a, while the attacker is probing the hash buckets, it does not disturb the lock hold times significantly and stays within the threshold limits. Trāṭr views this behavior as normal and does not flag it as an attack. Only when the attacker parks thousands of threads on the hash bucket, both the LCS and HSUA conditions meet, and Trāṭr flags it as an attack. Due to the aggressive probing used by Trāṭr, the attack is detected immediately within 1 second of when the attack starts when the synchronization stalls cross the threshold limits.

For the IC benchmark shown in Figure 5.10b, Trāṭr detects an attack within 1 second of when the attack starts. The probing window size expands dynamically whenever Trāṭr identifies that the synchronization



(a) Futex table attack



(b) Inode cache attack

Figure 5.10: **Performance of Trātṛ components.** *Throughput timeline for the futex and inode cache attacks explaining the importance of detection, prevention and recovery. We also show the timeline for Trātṛ-TDP. Trātṛ-TDP denotes the kernel version that has the tracking, detection and prevention mechanisms enabled. (a) For Trātṛ-TDP, as there is no recovery mechanism enabled, the FT benchmark observes a significant drop in the performance. (b) However, for IC benchmark, the prevention mechanism prevents the attacker from expanding the hash bucket leading to similar performance as Trātṛ.*

stalls are beyond the threshold limit. Therefore, within the same probing window itself, the attack is flagged.

On flagging an attack, Trātṛ quickly identifies the attacker by traversing the hash bucket, checking the user information stamped on the objects, and passing on the attacker's information to the prevention and recovery mechanisms. One thing to note here is that the attacker continues with the attack once the prevention window ends. Once the attacker tries to expand the hash bucket, Trātṛ will detect the attack and initiate prevention and recovery mechanisms.

5.3.2.2 Prevention

We now discuss how the prevention mechanism helps in preventing future attacks and throttle an ongoing attack. Trātṛ rate limits the attacker by stalling new object allocation until the prevention window expires. Trātṛ adjusts the prevention window size depending on how frequently a user is flagged as an attacker to slow down the attacker.

As seen in Figure 5.6, Trātṛ detects the inode cache attack early in the prepare phase of the attack, restricting the attacker's capability to extract the superblock. This helps to prevent future attacks as without knowing the superblock address, it is hard to launch a synchronization attack.

When the attacker already knows the superblock address, Figure 5.10b shows the timeline of the victim's throughput. As the inode cache attack is ongoing, the attacker can launch an attack immediately after the prevention window is over. The regular spikes in the throughput timeline indicate the attacks. Trātṛ learns the attacker's behavior and grows the prevention window accordingly to ensure that the attacker is further slowed down to prevent any impact on the victim's performance. The time gap between throughput drops roughly doubles every time, and at the end of the experiment, the prevention window size has grown to 216 seconds. Thus, Trātṛ can help in limiting an ongoing attack.

To understand the importance of prevention and recovery and their dependence on each other, we conduct another experiment where we disable the recovery mechanism and run the FT and IC benchmark (Trātṛ-TDP). Figure 5.10a shows the result for the FT benchmark. For Trātṛ-TDP, the victim's performance stays poor as the victims continue to traverse the entries allocated by the attacker in the hash bucket. Even though the attacker has turned passive, the expanded data structure continues to stay. Thus, for framing attacks, relying on the prevention mechanism alone is not enough. Recovery is needed to restore the data structure to a pre-attack state and recover baseline performance.

On the other hand, for Trātṛ-TDP, the IC benchmark observes similar performance to Trātṛ. The reason for such performance is that Trātṛ keeps growing the prevention window size. Moreover, as the victim does not access the targeted hash bucket frequently, there is minimal performance impact. However, as the hash bucket in the inode cache still has the entries allocated by the attacker, whenever the attacker again tries to add a few entries to the hash bucket, Trātṛ detects the attack and again initiates the prevention mechanism.

5.3.2.3 Recovery

Recovery is important to restore performance after an attack. Even though both the inode cache and futex table are hash tables, their recovery is different as they are designed for different purposes. We show that for different recovery solutions, Trātṛ can bring performance back to normal by quickly recovering after an attack.

For the inode cache, the recovery evicts all the entries belonging to the attacker. In Figure 5.10b, we observe that post-recovery, the performance goes back to baseline levels. With the preventive measures in place, the recovery completes quickly. For our experiments, the time to perform recovery remains around 4-5 milliseconds. Furthermore, the amount of

work done to recover is fixed as the attacker can only inflict damage from the start of the attack to when an attack is detected.

For the futex table attack, Trātṛ recovers by isolating the attacker from the victims. We consistently see that the recovery procedure of isolating the attacker completes within 50-70 milliseconds. Furthermore, as the attack is detected immediately after it is launched, the attacker cannot expand the hash bucket extensively, helping the recovery complete faster.

Without the prevention measures, the attacker can continue to expand the data structure holding the synchronization primitive. Under such circumstances, as the recovery procedures also acquires the synchronization primitives, longer lock hold times will stall the recovery making the recovery mechanism a victim. Thus, preventive measures are necessary for faster recovery.

Results summary. To summarize the results, we show that Trātṛ is effective, efficient, and responsive to detect and mitigate an attack. As Trātṛ can mitigate the attack immediately, the performance impact of the attack on the victim is negligible for both the recovery solutions.

5.3.3 Overhead

We now show the impact due to the overhead introduced by Trātṛ. We will be discussing two aspects of overhead – the performance overhead and the memory overhead.

5.3.3.1 Performance Overhead

Within the performance overhead, there are two aspects to look for - one due to the tracking mechanism and another due to the introduction of kernel threads that execute the detection, prevention, and recovery mechanisms.

Application	Description
CouchDB	CouchDB is a NoSQL database with support to replicate data across distributed servers for redundancy. It is a type of document database, stores data in JSON format.
Cassandra	Cassandra is a NoSQL database designed to run on "commodity" servers. It is a type of wide-column database.
LevelDB	LevelDB is a simple key-value NoSQL database with persistence storage.
RocksDB	RocksDB is a key-value NoSQL database, built on top of LevelDB. It is designed to take advantage of fast IO devices such as Flash drives.
InfluxDB	InfluxDB is a time-series database.
SQLite	SQLite is a weakly typed RDBMS system.
Darktable	It is an open-source photography software to manage RAW photos. The benchmark exploits its compute requirement to stress the system.
Kripke	Kripke is an Sn particle transport code; it is used to study the impact of various factors such as data layout, programming paradigm, and architecture on the performance of Sn transport
RAR	RAR is a file compression/archive utility.
MNN	Mobile Neural Network, developed by Alibaba, is a lightweight and efficient neural network framework.
NCNN	NCNN, developed by Tencent, is an optimized neural network framework.

Table 5.3: **Applications used for studying overhead.** *List of the applications that are part of the Phoronix test suite that we use to understand the overhead in Trāṭr.*

Benchmark	Description
Apache Benchmark	It is one of the standard benchmarks to test the performance of HTTP web servers.
Blogbench	Blogbench re-creates file server load by stressing the underlying filesystem. This benchmark imitates load generated on a blogging site while adding, modifying, or reading content.
Apache Siege	Siege is a tool to perform a load test on HTTP servers by making concurrent connections. We are spawning 500 connections to imitate a real-world scenario.
Dbench	The benchmark stresses the filesystem by spawning concurrent clients generating IO workload.
IOR	IOR is a parallel IO benchmark, with a particular focus on HPC workload. It depends on MPI for synchronization.
OSBench	OSbench is a set of micro-benchmarks designed to measure OS performance by creating files, threads, processes, and memory allocations.
Intel MPI Benchmark	It is a collection of MPI benchmarks.
Neatbench	Neatbench benchmark measures the system performance by executing the Neat video render program.
FinanceBench	Finance benchmark tests the system performance by running different financial applications.

Table 5.4: **Benchmarks used for studying overhead.** *List of the benchmarks that are part of the Phoronix test suite that we use to understand the overhead in Trāṭr.*

Tracking mechanism overhead. The tracking mechanism is the only code that is executed in the critical path. The other three mechanisms – detection, prevention, and recovery, happen in the background by kernel threads. Therefore, it is important to understand the overhead Trāṭr introduces to track the objects.

To understand the impact of tracking, we use the Phoronix test suite – a free and open-source benchmark suite that supports more than 400 tests [126]. The test suite allows to execute tests and report the results in an automated manner. For our experiments, we run the Docker container that is provided by the Phoronix test suite.

We run the experiments on a variety of applications and benchmarks provided by the Phoronix test suite. Table 5.3 shows the list of all the applications and the description of the applications that we use for experimentation. Table 5.4 shows the list of all the benchmarks and the description of the benchmarks.

We choose these applications and benchmarks as they stress the underlying kernel. As they interact with the kernel, they will have to allocate the kernel objects, stressing the tracking mechanism. We run the container with unrestricted CPU access as multiple objects will be created concurrently on all CPUs stressing the tracking mechanism's parallelism. For the majority of the listed applications and benchmarks, the test suites run tests three times. We use the average of these three tests or more for the comparison.

As Trāṭr updates its accounts on object allocation and freeing, the performance overhead is paid only at the time of allocation or freeing. While accessing objects, there is no overhead. We used a special version of Trāṭr called Trāṭr-T that enables tracking for all the slab-caches to measure this performance difference. We compare Trāṭr-T's performance with the Vanilla kernel's performance.

Table 5.5 shows the performance of Trāṭr kernel compared to the

Application	Test Detail	Relative Trāṭr Performance
CouchDB	Insertions	1.20%
Cassandra	Mixed: Write and Reads	0.92%
LevelDB	Random Read	5.32%
	Fill Sync	-3.46%
	Overwrite	-2.96%
	Seek Random	0.85%
	Sequential Fill	-2.49%
RocksDB	Sequential Fill	-4.16%
	Random Fill Sync	-1.53%
	Read While Writing	0.41%
InfluxDB	Concurrent write	-0.25%
SQLite	Insertions	-3.33%
Darktable	"Boat" test using CPU only	-3.05%
Kripke	Equation solver	-1.90%
RAR	Compress Linux kernel	-2.64%
Mobile Neural Network	Inference on inception-v3 model	-0.11%
NCNN	Inference on regnety_400m model	-1.31%

Table 5.5: **Performance overhead study for applications.** *Comparison of performance for the various applications for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches relative to Vanilla kernel.*

Benchmark	Test Detail	Relative Trāṭr Performance
Apache Benchmark	Static Web Page Serving	-0.58%
Blogbench	Read	-0.13%
	Write	-9.67%
Apache Siege	Concurrent connection on web server	-2.26%
Dbench	Concurrent clients doing I/O	-0.44%
IOR	Parallel I/O tests with 1024 block size	-1.23%
OSBench	Create Files	-7.22%
	Create Threads	-2.25%
	Launch Programs	-6.14%
	Create Processes	-9.54%
	Memory Allocations	0.31%
Intel MPI Benchmark	PingPong Test	-0.03%
Neatbench	Neat Video render using CPU	5.88%
FinanceBench	Bonds OpenMP Application	-1.19%

Table 5.6: **Performance overhead study for benchmarks.** *Comparison of performance for the various benchmarks for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches.*

Vanilla kernel for the applications used for experimentation. The last column in the table shows the difference in the performance. For the majority of the applications, Trāṭr's performance is between 0-4% less than the Vanilla kernel's performance. The NUMA-aware hash table used to store the user's accounting information helps keep the performance difference minimal. We also do observe a slight improvement in the performance of a few applications. However, we do not have a specific reason for an increase in the performance compared to the Vanilla kernel.

Table 5.6 shows the performance of Trāṭr kernel compared to the Vanilla kernel for the benchmarks used for experimentation. For the majority of the benchmarks, the performance difference is between 0-5% less than the Vanilla kernel's performance. However, for a few benchmarks, the decrease in the performance is slightly higher. This is because these benchmarks try to create hundreds of kernel objects in a short period, overwhelming the hash table used to track each user's accounting information. For example, the OSBench benchmark creates threads and processes in a loop stressing the tracking mechanism. However, we believe that very few real-world applications create thousands of threads or processes in such a short period.

Kernel threads overhead. The kernel threads probe the synchronization primitive to detect an attack. On detecting an attack, these threads will initiate the prevention and recovery mechanisms. Even though these threads run in the background, there is still a possibility that these kernel threads may interfere with the application's threads and compete for the CPU.

To understand the impact of these kernel threads, we use the same Phoronix test suite for experimentation. We use three test applications described in Table 5.7 to measure the impact of the kernel threads.

We run four containers where the same test application is executed, and each container is allocated 8 CPUs. By running the same test in all

Application	Description
N-Queens	This OpenMP version solves the N-queens problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. The board problem size is 18.
CP2K Molecular Dynamics	CP2K is an open-source molecular dynamics software package focused on quantum chemistry and solid-state physics used for performing atomistic simulations of solid-state, liquid, molecular, and biological systems.
Primesieve	Primesieve generates prime numbers using a highly optimized sieve of Eratosthenes implementation. Primesieve benchmarks the CPU's L1/L2 cache performance.

Table 5.7: **Applications used to study kernel threads overhead.** *List of the applications that are part of the Phoronix test suite used for measuring the impact of kernel threads.*

four containers, we can easily measure the variation in the performance caused by the kernel threads. Additionally, different tests run for a different time duration, and hence it will be hard to find tests that run for the same amount of time necessary for our experiments. Moreover, the test suite does not allow running the tests for a specific time duration.

For all the three test applications, we find that the performance difference between maximum runtime for Trātṛ and minimum runtime for Vanilla kernel to be somewhere around 1-1.5%. This performance difference corroborates with the design strategy discussed in Section 5.2.3.2. On average, the kernel threads will probe the synchronization primitives every 12.5 milliseconds. It takes about 120 microseconds to complete one probing. Therefore, within one second, the total CPU time used by the kernel threads is roughly ten milliseconds which is 1% of 1 second.

Application	Active Slab Size in Vanilla (in MB)	Active Slab Size in Trāṭr (in MB)	Slab Overhead in Trāṭr (%)
CouchDB	350.05	360.19	2.9 %
Cassandra	210	230	9.52%
LevelDB	66	71	7.58%
RocksDB	91.52	95	3.8 %
InfluxDB	62.96	74.33	18.06%
SQLite	92.49	101.27	9.49%
Darktable	115.06	116.77	1.49%
Kripke	46.49	50	7.55%
RAR	77.16	80.26	4.02%
MNN	102.87	110.81	7.72%
NCNN	94.71	102.59	8.32%

Table 5.8: **Memory overhead study for the applications.***Comparison of the memory overhead for various applications for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches. The numbers in the bracket in the last column show the % increase in the total memory allocated to all the slab caches.*

5.3.3.2 Memory Overhead

Trāṭr uses a hash table with 32 buckets per NUMA node to track the memory allocations per user for a slab cache. Our machine has two NUMA nodes; adding the hash table to the slab-cache structure increases its size by 544 bytes. Therefore, to support thousands of slab-caches, the total memory overhead is less than 1 MB. With more NUMA nodes, the total memory overhead may be higher.

Trāṭr also adds 4 bytes of extra memory allocated per object to stamp the user-id. As the Linux kernel uses slabs for object allocation, increasing the object's size by 4 bytes will reduce the total objects allocated per slab. Using `slabtop` command [40], for an idle system having 150 slab-caches initialized, 3.8% more memory is allocated to slab caches for Trāṭr-T (219 MB) than the Vanilla kernel (211 MB).

Benchmark	Active Slab Size in Vanilla (in MB)	Active Slab Size in Trāṭr (in MB)	Slab Overhead in Trāṭr (%)
Apache Benchmark	73.6	78.7	6.93%
Blogbench	264.85	284.51	7.42%
Apache Siege	38	39.4	3.68%
Dbench	170.03	189.96	11.72%
IOR	62.89	74.33	18.19%
OSBench	87.17	98.86	13.41%
Intel MPI Benchmark	54.69	58.9	7.7 %
Neatbench	28.71	31.94	11.25%
FinanceBench	72.61	77.09	6.17%

Table 5.9: **Memory overhead study for the benchmarks.***Comparison of the memory overhead for various benchmarks for the Vanilla kernel and Trāṭr-T with just tracking enabled for all the slab-caches. The numbers in the bracket in the last column shows the % increase in the total memory allocated to all the slab caches.*

We also monitor the slab cache usage for the 20 tests to understand the tracking mechanism's overhead. Using slabtop command, we measure and compare the total memory allocated to all the slab caches for Trāṭr and Vanilla. Table 5.8 and Table 5.9 shows the memory overhead caused due to the addition of 4 bytes of extra memory to each object.

We notice that the applications and benchmarks that regularly creates object are more likely to observe a higher memory overhead. On the other hand, applications and benchmarks that create few objects and use the same objects several times amortize the cost of the object tracking leading to lower memory overhead. We believe that given the amount of main memory available today, a few MB of extra memory used by slab-caches may not hurt performance.

Application	Workload
DBENCH	32 threads executing client loadfile workload
UpScaleDB	ups_bench –inmemorydb –num-threads=64
Exim	Mosbench workload using 8 clients

Table 5.10: **Real-world scenarios study with three real-world applications.** *List of the real-world applications and their workloads used for understanding how Trāṭr performs in real-world scenarios.*

Container	Scenario 1 & 2		Scenario 3	
	Futex table & Inode cache attack		Multiple attacks	
	CPU	Memory	CPU	Memory
Exim	8	16 GB	8	16 GB
DBENCH	8	16 GB	8	16 GB
UpscaleDB	8	64 GB	8	64 GB
Attacker	1	8 GB	4	8 GB

Table 5.11: **Real-world scenario description.** *List of three real-world scenario summary and resource allocation to each container in each of the scenario.*

5.3.4 Real-World Scenarios

We now demonstrate how using Trāṭr in real-world scenarios can prevent synchronization primitives from turning adversarial. We focus on three different applications – the Exim mail server, UpScaleDB, and DBENCH benchmark. Using three different scenarios, we explain the importance of Trāṭr. Table 5.11 summarizes each scenario. We run the workload as described in Table 5.10 for 300 seconds.

Scenarios 1 and 2 deals with the futex table and inode cache attack and are an extension of the attacks discussed in Section 3.2.3. Instead of running a single victim, we run more victims to illustrate the real-world scenario where multiple containers run on a single physical machine.

The performance comparison of Vanilla and Trāṭr with and without futex table attack is shown in Figure 5.11a. We use the throughput of the

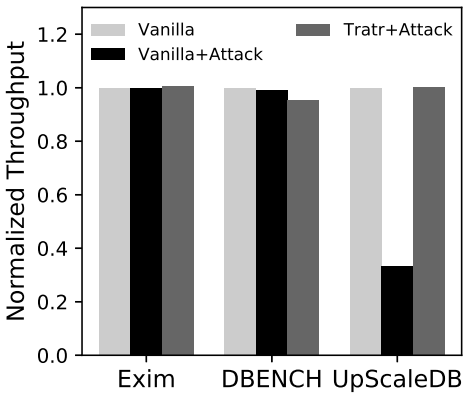
Vanilla kernel as the baseline to calculate the normalized throughput. As UpScaleDB relies on Pthread mutex lock [123]; the futex table attack impacts its performance. Figure 5.11b shows the performance comparison for the inode cache attack. As Exim and DBENCH use inode cache for file creation and deletion, the attack impacts their performance with Vanilla kernel. For both the scenarios, Trātṛ can detect and mitigate the attack with minimal impact.

In Scenario 3, an attacker launches both the futex table and inode cache attack simultaneously. This experiment shows how Trātṛ can handle simultaneous attacks without impacting victims' performance. Figure 5.11c shows that all three victims observe poor performance in Vanilla kernel. On the other hand, Trātṛ detects both attacks and employs different recovery solutions to mitigate both attacks without impacting the victim's performance. Thus, Trātṛ is not limited to just detecting and mitigating a single attack from an attacker. Trātṛ can tame the attackers that try to turn the synchronization primitives adversarial.

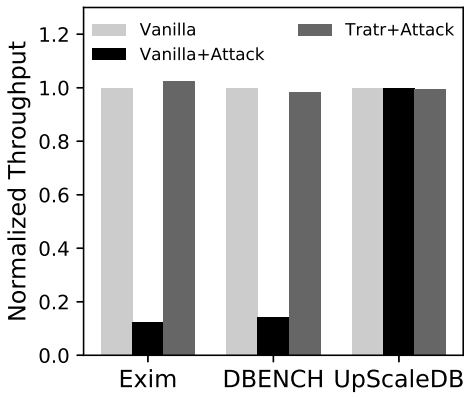
Cost of the attack. For Scenario 1 & 2, we use 1 CPU, and for Scenario 3, we use 2 CPUs to launch attacks to run two attacks. Even with 1 CPU, the attacker can generate synchronization interference leading to poor performance. Thus, to launch either synchronization or framing attacks, the cost associated with the attacks is minimal. One should note here that with more resources, a more severe attack can be launched.

Economic impact. We already discussed earlier the economic impact on the victims due to the synchronization and framing attack. For scenario1 and scenario3, for the victim (UpScaleDB) of the futex table attack, we observe that the victim endup up increasing the CPU usage by around 2.25 to 2.4X compared to the baseline. On the other hand, for Trātṛ, there is an increase in the total CPU usage by 0.5% only.

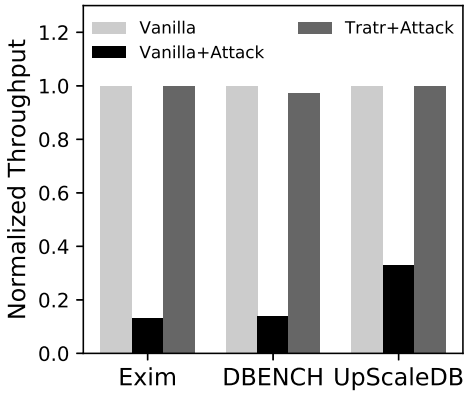
For the inode cache attack in Scenario 2 and 3, while the Exim Mail Server spends around 31-32.45% of the total runtime, DBENCH spends



(a) Scenario 1 - Futex table attack



(b) Scenario 2 - Inode cache attack



(c) Scenario 3 - Multiple attacks

Figure 5.11: **Performance comparison of the various applications across different scenarios.** *Performance comparison of Vanilla and Trātṛ with and without attack when subjected to different attacks. (a) & (b) shows the performance when multiple applications run within a single machine for futex table and inode cache attack. (c) shows the ability of Trātṛ to handle simultaneous attacks.*

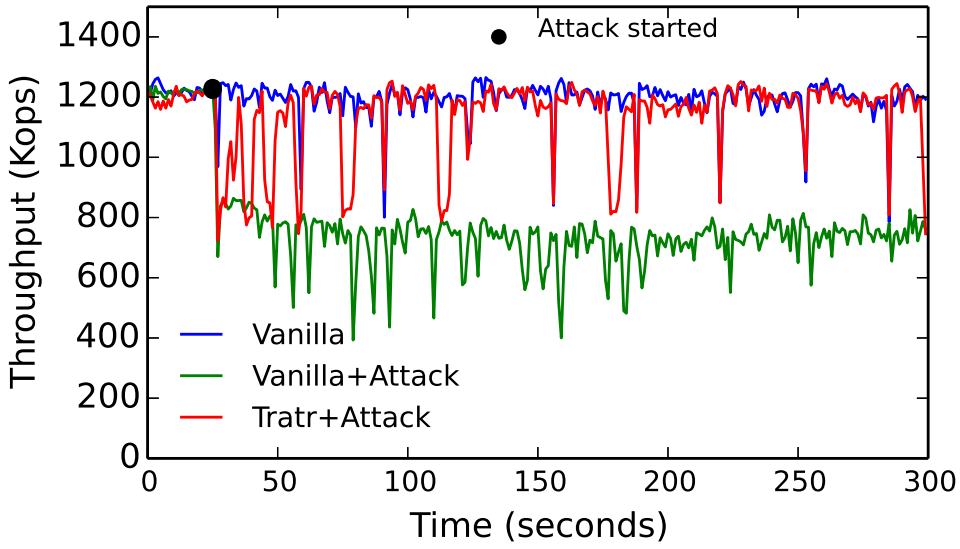


Figure 5.12: **Performance comparison of the Exim mail server when under dcache attack.** *Performance comparison of Vanilla and Trātṛ with and without attack when subjected to directory cache attack. Timeline of the throughput showing the impact on the throughput due to the attack for the Vanilla and Trātṛ kernel.*

around 45-46.73% of the total runtime waiting to acquire the inode cache lock. This shows that all the victims will not only suffer due to poor performance, but they will end up having an economic impact too.

5.3.5 Adding Directory Cache to Trātṛ

One of our goals is to ease the process of adding a new data structure to Trātṛ. We choose the dcache to understand the effort needed to implement the CSP & TCA checks and recovery. We first set the tracking flag for the dcache slab-cache to enable tracking. Dcache uses RCU and bit-based spinlocks to support concurrent access. We focus on the RCU because the spinlock's critical section is small and will be hard to turn adversarial.

CSP checks involve probing the grace period size by measuring the time to complete the `synchronize_rcu()` call until the probing window expires. The probing threshold is set to 15 milliseconds which is twice the size of the CPU time slice. The TCA check walks the hash bucket having the most entries. As the dcache is used for performance purposes, the recovery procedure evicts all the attacker's entries. We re-use existing code to evict all the attacker's entries from all superblocks. In total, we write 120 lines of new code to add the dcache to Trātṛ.

To test the new code changes, we run the same directory cache attack described in Section 3.2.3. The simulated attack targets a single hash bucket and creates thousands of negative entries to increase the RCU grace period . Figure 5.12 shows the result for the Vanilla and Trātṛ kernel with an attack. We also show the performance of the Vanilla kernel with an attack from Figure 3.6. We observe that Trātṛ can detect and mitigate the attack quickly. With a small prevention window initially, the attacker can launch new attacks immediately after the prevention window expires. As time passes, Trātṛ increases the prevention window size, reducing the number of throughout drops.

One point to note is that many subsystems use the RCU subsystem within the kernel; it is possible to launch the same attack by targeting another data structure like dcache. So even though Trātṛ can detect and mitigate the directory cache attack, there is still a possibility that an RCU attack can be launched by targeting another data structure.

5.3.6 False Positives

One of our design goals is to have low false positives so that Trātṛ does not detect the victims as an attacker and unnecessarily penalize them by preventing them from creating more objects and initiating recovery leading to poor performance. As Trātṛ relies on threshold limits to detect if a synchronization primitive is under an attack, badly configured applications

or stress testing scenarios may be detected as an attack.

We conduct a false positives study to identify how many times Trātṛ flags a victim as an attacker. We use the same 20 applications and benchmarks used in Table 5.3 and Table 5.4 and club five applications onto a single container to run random workloads. In total, we create four containers using these 20 applications and benchmarks. We allocate 8 CPUs to each container and then randomly run the Phoronix test suite's stress tests to stress the system. As the stress tests stress the system, we believe there may be scenarios where the threshold limits may be crossed, making Trātṛ to flag such scenes as an attack. We continue to run the stress tests for 24 hours duration.

During the 24 hour duration, we did not find any situation where Trātṛ flags the victims(the applications and the benchmarks) as an attacker. As discussed in Section 5.2.3.2, the threshold limits are carefully calculated, keeping in mind the heavy-contention criteria and the number of CPUs in the machine. For a different machine configuration, the threshold limits might change. Automatically choosing the threshold values is an interesting avenue to explore for future work. Thus, stressing the system still is not able to breach the threshold limits.

During our internal testing, we deliberately configured the experiment badly to see if we can make the filebench-webserver [149] workload flag as an attack. As expected, we observe that Trātṛ wrongly identifies the workload as a futex table attack a few times during the span of the experiment. As the workload is not aggressively accessing the futex table, the prevention window size never grows quickly. More so, during the prevention window, the workload does not create more threads, so it does not have to stall, leading to negligible performance reduction (less than 1%).

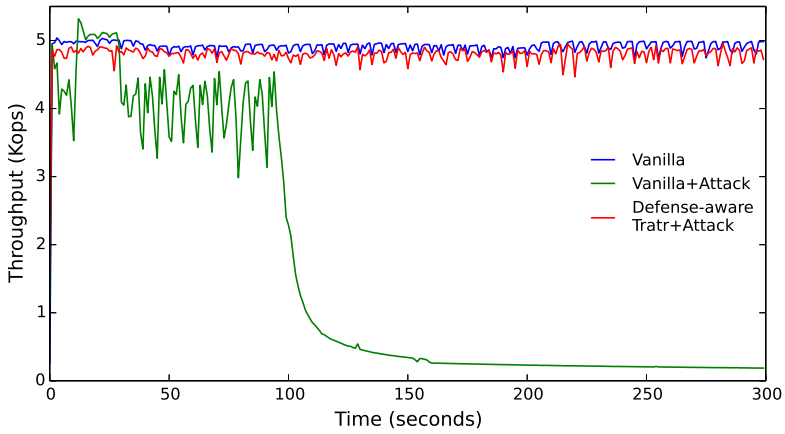
5.3.7 False Negatives

Along with reducing the false positive rate, our goal is to detect as many attacks as possible, reducing the false-negative rate. As Trātṛ relies on threshold limits to detect an attack, any attack that stays within the threshold limits is likely to be not detected as an attack. As seen in Figure 5.3, a defense-aware attacker knowing the threshold limits is highly likely to start an attack such that Trātṛ cannot detect an attack. While probing the locks, Trātṛ will never find any probe exceeding the threshold limit and hence will not flag an attack.

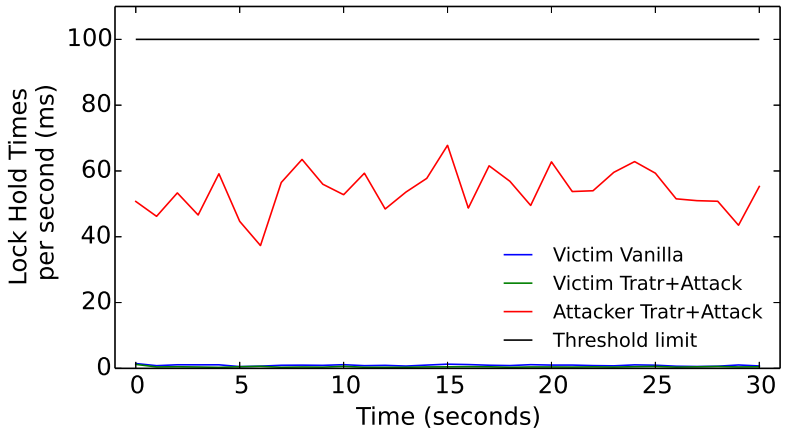
We now describe two scenarios – one for the inode cache attack and another for futex table attack where Trātṛ cannot detect an attack. We also discuss the performance implications when Trātṛ is not able to detect the attack.

Defense-aware inode cache attack. In this attack, the strategy of the attacker is to expand the hash bucket in such a way that inserting or accessing the entries in the hash table does not exceed the threshold limits. Therefore, the attacker first creates thousands of entries which will increase the critical section size impacting the victims. When the attacker reaches the threshold limits, it stops expanding the hash bucket and then deletes those entries. The attacker continuously creates and deletes the entries in a loop. More so, the attacker also needs to care about the detection window size. Trātṛ moves into an aggressive mode while probing if it finds that the synchronization stall is more than the threshold even once.

To illustrate the scenario, we again use the same IC benchmark used earlier as the victim. Figure 5.13 shows the victim's throughput timeline for the attack. For comparison purposes, we show the victim's performance on the Vanilla kernel with and without a full-fledged attack to highlight the performance impact that a defense-aware attacker can make. We use the results of the experiments conducted in Section 5.3.1.1.



(a) Throughput timeline for the inode cache attack highlighting the victim's performance



(b) Lock hold times for the inode cache attack.

Figure 5.13: **Performance comparison of victim when a defense aware attacker launches inode cache attack.** *(a) Timeline of the throughput showing the impact on the throughput due to the defense-aware attacker. (b) Lock hold times comparison to highlight how the defense-aware attacker remains under the threshold limits to evade detection.*

While running the experiment, we also measure the lock hold times every second. Figure 5.13b shows the maximum lock hold times by the attacker. We observe that the lock hold times are always less than the threshold limits showing how a defense-aware attacker can dodge Trātṛ and remain undetected.

However, we observe that a defense-aware attacker is not able to cause much damage to the victim. As the attacker cannot acquire the lock for more than threshold limits to avoid detection, the victims do not have to wait longer to acquire the inode cache lock. Moreover, due to the random probing window size and frequency, the attacker cannot continuously create and delete files within a single probing window. If the attacker does breach the threshold limits several times within a probing window, Trātṛ will detect the attack. Thus, the victim will not observe poor performance or denial-of-services when a defense-aware attacker will launch the inode cache attack.

Defense-aware futex table attack. In this attack, we launch a synchronization attack on the futex table where the attacker's strategy is to actively participate in the lock acquisition process leading to lock contention. The attacker does so by identifying the target bucket and then creating tens of threads that will continuously call futex syscalls to traverse the hash bucket. To remain undetected, the attacker does not add entries to the target hash bucket so that even if the threshold limit is crossed, a victim may be falsely implicated as an attacker when Trātṛ traverses the hash bucket while performing the TCA check.

We use the FT benchmark used earlier as the victim. On launching the defense-aware attack, we are unable to force Trātṛ to flag the victim as an attacker wrongly. As the hash bucket is not that long enough, even with 24 attacker threads, there is not enough contention that Trātṛ while probing the hash table lock observes that the synchronization stall is more than the threshold limit. Furthermore, as the attacker is not aware of the

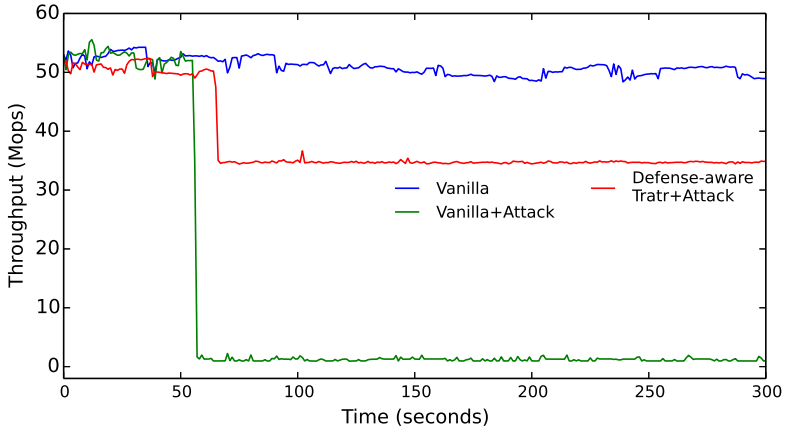


Figure 5.14: **Performance comparison of victim when a defense aware attacker launches futex table attack.** *Timeline of the throughput showing the impact on the throughput due to the defense-aware attacker.*

total entries of the victim in the hash table, the attacker cannot expand the hash table bucket such that the victim's total number of entries is more than that of the attacker's and still manages to cross the threshold limits.

Even though the attacker cannot falsely implicate the victim as an attacker, we observe that the attacker can still reduce the victim's performance by around 20%. Figure 5.14 shows the throughput timeline of the victim. We also observe that with lesser threads, the impact of the attack reduces. Hence, the impact of such a defense-aware attack will depend on the resources that the attacker has. With more CPUs, the attack's impact will be higher compared to fewer CPUs. For comparison purposes, we also show the performance of the FT benchmark on the Vanilla kernel with and without a fill-fledged futex table attack described earlier in Section 5.3.1.2.

Even though the impact of the defense-aware attack is not as severe as the full-fledged attack, we believe that Trātṛ should be able to detect such scenarios and flag them as an attack. One possible way to reduce

the performance impact is to use Scheduler-Cooperative Locks(SCLs) instead of the traditional locks. As SCLs guarantee lock usage fairness, the attacker, in this case, will be penalized heavily for dominating the lock usage allowing the victim enough opportunity to acquire the lock.

5.4 Limitations

We now discuss two limitations of Trātṛ. Firstly, as there is no distinct boundary defining a particular behavior as a minor performance inconvenience or a significant performance problem, Trātṛ uses probing thresholds to detect an attack. A defense-aware attacker may be able to stay within these threshold boundaries and remain undetected. Under such a condition, the victim may continue to observe minor performance issues where the attacker can elongate the critical section size to threshold values.

By lowering the threshold boundaries, Trātṛ can push the attack boundary lower. However, by doing so, Trātṛ may end up increasing the false positive rate. Our goal is to avoid false-positive cases as much as possible. By replacing the existing mutual exclusion locks with SCLs, minor performance inconvenience can be avoided by guaranteeing lock usage fairness.

Secondly, an attacker can use other services available in the operating system to expand a data structure making the service accountable for its size. For example, the attacker can ask the print spooler to load files whose hash values map to a single bucket for printing. Trātṛ will treat the print spooler as the one who created the inodes. In the worst case, Trātṛ might tag the print spooler as the attacker.

5.5 Summary & Conclusion

A majority of the enterprises are moving to the shared infrastructure provided by cloud platforms for efficiency purposes. Ideally, we would like to consider processes as running in their own isolated and perfectly virtualized environment in such an environment. However, the reality is that modern applications and services run atop a concurrent shared infrastructure. In this shared infrastructure, without strong performance isolation, the behavior of one tenant can harm the performance of other tenants. As we showed, an attacker can monopolize the synchronization primitives used to build the data structures within the shared infrastructure leading to poor performance and denial-of-services attacks.

One might think that mitigating adversarial synchronization can be solved through code refactoring or avoidance techniques like universal hashing or balanced trees. Unfortunately, doing so is not always straightforward; shared infrastructure systems are often highly complex, and rewriting them is no simple matter. For example, the Linux kernel comprises hundreds of data structures protected by thousands of synchronization primitives. As a result, identifying and replacing vulnerable data structures is not an easy task.

In this chapter, to remedy adversarial synchronization, we introduced Trāṭr– a Linux kernel extension to defend against adversarial synchronization. Trāṭr can detect attacks within seconds and instantaneously recover from the attack, bringing the victims' performance to baseline levels. In addition, Trāṭr is light-weight, imposes minimal overhead for tracking, and adding new data structures to Trāṭr is easy and flexible. The source code for Trāṭr, the attack scripts, and the experimental setup is open-sourced and can be accessed at https://research.cs.wisc.edu/adsl/Software/.

We strongly believe that the data structures and the synchronization primitives protecting these data structures that are part of the shared in-

frastructure should be considered resources. In doing so, we can isolate the shared infrastructure properly to avoid fairness and starvation-related problems. By discussing the design and evaluation of Trātṛ, we showed that the fairness and starvation problems can be solved.

6 Related Work

In this chapter, we discuss other pieces of work that are related to this dissertation. We start by discussing how locks are different than other resources and explain the importance of lock usage fairness compared to lock acquisition fairness in Section 6.1. We then describe other problems that are related to the scheduler subversion problem and how prior work acknowledges the interaction between schedulers and locks in Section 6.2. We compare the adversarial synchronization problem and algorithmic complexity attacks and their solutions in Section 6.3. We describe how Scheduler-Cooperative Locks fit in the five locking algorithms and compare them with the existing state-of-the-art in Section 6.4. Lastly, we compare the design of Trāṭr with other existing solutions that detect and prevent algorithmic complexity attacks in Section 6.5.

6.1 Lock Usage Fairness

Researchers and practitioners have developed many techniques to ensure resource isolation and fairness guarantees. The majority of previous work has focused on CPU [47, 72, 112], memory [112, 158], storage [100, 143], and network isolation [81]. However, to the best of our knowledge, there has not yet been work ensuring lock usage fairness. One important difference between locks and other resources is preemptability. Once the

lock is acquired, ownership cannot be revoked until the lock is voluntarily released; our work presumes the continued importance of such locks in concurrent systems. Therefore, the solutions proposed for other resources cannot apply to non-preemptive resources like locks. We believe that there are other non-preemptive resources like GPU [24] that can also benefit by applying the idea of opportunity-based fairness.

Some existing locks, like ticket lock, MCS, offer fairness by ensuring that no process can access a lock twice while other processes are kept waiting [138]. This type of fairness guarantees that locks are *acquired* fairly, thereby not starving any waiting thread. However, the lock acquisition fairness does not consider the critical section time and is more likely to create an imbalance when critical section time varies. Therefore, we are adding a new dimension to the existing fairness property of the locks by introducing lock usage fairness and lock opportunity.

6.2 Scheduler subversion

Many studies have been done to understand and improve the performance and fairness characteristics of locks [36, 50, 70]. However, the authors do not consider lock usage fairness as we propose herein. Guerraoui et al. [70] do acknowledge that interaction between locks and schedulers is important. This thesis shows how locks can subvert the scheduling goals; SCLs and schedulers can align with each other.

The closest problem related to the scheduler subversion problem is the priority inversion problem [140] where a lower priority process blocks a higher priority process. The scheduler subversion problem can occur with any priority thread, and as we have shown, it can also happen when all the threads have the same priority. We believe that to prevent priority inversion, priority inheritance [140] should be combined with SCL locks.

Another interesting study compares various coarse-grained locks,

fine-grained locks, and lock-free methods to understand the throughput and fairness [36]. The authors define fairness by comparing a thread's minimum number of operations with the average number of operations across all threads. However, the authors do not consider lock usage fairness as we propose herein.

6.3 Adversarial Synchronization

At first glance, the problem of adversarial synchronization looks similar to algorithmic complexity attacks [46] as both of these deal with denial-of-services. The algorithmic complexity attacks so far focus on exhausting one or more CPUs in the system [1–9, 144]. However, when applications are run in a containerized environment where each container is allocated its own CPUs, there is less chance that algorithmic complexity attacks may impact other containers too. The isolation guarantees provided by the containers limit the impact of the algorithmic complexity attack to a single target container only.

On the other hand, adversarial synchronization targets the shared infrastructure like the underlying operating systems on which the containers run. Therefore, all the containers that interact with the operating systems will be impacted. When an attacker launches synchronization or framing attacks and targets the synchronization primitives, the attacker will force all the victims to either wait longer to access the synchronization primitives or force them to expand their critical section leading to poor performance or denial-of-services.

Algorithmic complexity attacks target the preemptive resources such as CPU, disk, or network; adversarial synchronization targets the non-preemptive resources like synchronization primitives. There have been multiple solutions proposed to either avoid or detect and prevent the algorithmic complexity attacks [41, 44, 46, 122, 138]. However, as we have

shown in Chapter 5, none of these solutions can solve the problem of adversarial synchronization.

6.4 Scheduler-Cooperative Locks

Locks have been split into five categories based on their approaches [71]: (i) flat, such as pthread mutexes [92], spinlocks [138], and many other simple locks [16, 48] (ii) queue, such as ticket locks [45, 102, 109, 109], (iii) hierarchical [37, 38, 56, 101, 131], (iv) load-control [53, 73], and (v) delegation-based [62, 74, 99, 119].

Our work borrows from much of the existing literature. For example, queue-based approaches are needed to control fairness, and hierarchical locks contain certain performance optimizations useful on multiprocessors. For example, we use the existing K42 lock implementation [138] to implement the u-SCL lock. By doing so, we were able to not only preserve the existing properties of K42 locks, but also able to add lock usage fairness.

Our approach shows that it is easy to expand the existing locking algorithms to add lock usage fairness. By adding the three components – lock usage accounting, penalizing threads depending on lock usage, and dedicated lock opportunity using lock slice, existing locks can be made to guarantee lock usage fairness. Additionally, the applications that use the locks will not have to be modified extensively.

Reader-writer locks have been studied extensively for the past several decades [32, 34, 54, 90, 95, 110, 111] to support fairness, scaling, and performance requirements. We also borrow the idea of splitting the counter into per-NUMA node counter to avoid a performance collapse across NUMA nodes [34].

Brandenburg et al. [32] present a phase fair reader-writer lock that always alternates the read and write phase, thereby ensuring that no starva-

tion occurs. Our approach for RW-SCL and the phase fair reader-writer lock does have certain properties in common. Like the phase fair lock, RW-SCL will ensure that read and write slices alternate. Furthermore, RW-SCL is flexible enough to assign a lock usage ratio to readers and writers depending on the workload. Again, by reusing existing reader-writer lock implementation, we show how lock usage fairness can be added by extending the current implementation.

We borrow the idea of lock cohorting to introduce the idea of lock slice [57]. To avoid excessive lock ownership migrations across NUMA nodes, cohort locks take turns to allow threads within a single NUMA node to acquire the lock. Thus, a lock slice is similar to taking turns. However, the lock slice is only allocated to a thread by considering the lock usage across all the participating threads.

The idea of a lock slice is also similar to the time slice used for CPU scheduling. A time slice is a short time that gets assigned to a process or thread for CPU execution. Similarly, a lock slice is a short time that gets assigned to a schedulable entity. A lock slice virtualizes the critical section to make the thread believe that it has the lock ownership to itself the way time slices virtualize the CPU and makes each thread believe that it has the CPU to itself. With lock slice and dedicated lock opportunity, even if the lock is free, other waiters will not be able to acquire the lock. This makes SCLs non-work-conserving compared to the traditional lock that are designed to be work-conservative.

Although less related, even delegation-based locks could benefit from considering usage fairness, perhaps through a more sophisticated scheduling mechanism of delegated requests. There is a possibility that the CPU scheduler may not be aware of a low-priority thread being delegated by high-priority threads. Such a scenario can also lead to the scheduler subversion problem, where the low priority thread is allocated more CPU than high priority threads.

Recently, Park et al. introduced the idea of Contextual Concurrency Control where the developers can tune the kernel synchronization mechanisms from userspace on the fly [120]. In addition, they illustrate that the scheduling subversion problem can be solved by allowing application developers to inject fairness-related code only when needed instead of enforcing it always. Although still a work in progress, it is an interesting approach where the injected code can adhere to the underlying CPU scheduling policy.

6.5 Trāṭr

Detection and prevention is another approach to tackle algorithmic complexity attacks. Khan et al. propose an alternative to randomization through regression analysis based model to prevent attacks [89].

Qie et al. propose an approach where they show that annotating application code can help detect resource abuse and initiate rate-limiting or dropping of the attackers [130]. Similarly, FINELAME also uses annotation and probing to detect attacks [51]. DDOS-Shield assigns continuous scores to user sessions, checks these scores to identify suspicious users, and prevents them from overwhelming the resources [132]. Trāṭr too looks for anomalous resource allocations by checking LCS and HSUA. Moreover, prevention is not enough to address framing attacks. No other approach considers synchronization as a resource, unlike Trāṭr.

Radmin learns and executes Probabilistic Finite Automatas offline of the target process of all the monitored resources and then performs anomaly detection by making sure that the target programs stay within the learned limits [58]. It will be an interesting avenue to explore building lock usage models and use them to detect attacks in Trāṭr.

A continuous effort is being made to reduce the security concerns posed by the use of containers [97, 103, 142, 145]. The majority of these

studies focus on the fact that containers should not access privileged services. However, as we show in this paper, an attacker can launch attacks by executing simple and unprivileged code. Therefore, approaches that propose to scan the layers of the container to detect vulnerabilities may not detect the attackers that can launch synchronization and framing attacks [78].

An effort is also being made to provide lightweight isolation platforms such Google gVisor [79] and AWS Firecracker [15] by moving the functionality out of the host kernel and into the guest environment. The expectation is that by relying less on the host kernel, the attack surface will reduce, thereby addressing the security concerns, including the adversarial synchronization problem. However, a recent study shows that despite moving much of the host kernel functionality into the guest environment, both Firecracker and gVisor execute more kernel code than native Linux [17].

7
Conclusions & Future Work

Synchronization primitives play a vital role in concurrent systems. Amongst all the synchronization primitives, mutual exclusion locks are the most widely used primitives as they provide an intuitive abstraction to guarantee correctness. Locks exhibit specific properties crucial to guaranteeing any concurrent system's correctness, fairness, and performance requirements.

In the first part of the dissertation, we introduced and studied a new property - lock usage. We showed how with varying critical section sizes in a shared environment, unfair lock usage could lead to performance and security problems. In a shared environment, unfair lock usage can happen naturally in benign settings when a data structure grows while executing a workload. Conversely, in a hostile setting, a malicious actor can exploit the vulnerable data structures to dominate the lock usage leading to poor performance and denial-of-service.

In a benign setting, we introduced a new problem of scheduler subversion where a dominant thread spending more time in the critical section determines the proportion of the CPU each thread obtains instead of the CPU scheduler. In the hostile setting, we introduced a new class of performance attacks - synchronization and framing attacks where a malicious actor can artificially introduce lock contention leading to longer wait times and force victims to spend more time in the critical section.

In the second and third parts of the dissertation, we presented solutions to address the scheduler subversion and adversarial synchronization problems. To handle scheduler subversion, we introduced a new metric – lock opportunity to measure lock usage fairness and designed Scheduler-Cooperative Locks that can guarantee lock usage fairness while aligning to the CPU scheduling goals. To tackle the adversarial synchronization problem, we introduced Trātṛ, a Linux extension that can effectively detect and mitigate synchronization and framing attacks.

In this chapter, we first summarize each part of this dissertation in Section 7.1 and present the various lessons we learned through the course of this dissertation work in Section 7.2. We discuss some directions for future work in Section 7.3 and lastly conclude with closing words in Section 7.4.

7.1 Summary

We now provide a summary of the three parts of this dissertation.

7.1.1 Lock Usage

One of the most crucial properties locks exhibit is fairness, as it guarantees a static bound before a thread can acquire a lock and make forward progress. The static bound is defined by the order in which the threads acquire the locks, thereby not allowing one thread to acquire the lock again until other waiting threads are given a chance to acquire the lock.

In the first part of the dissertation, we showed that another crucial property that we call lock usage is missing in the previous approaches. Lock usage is the amount of time spent in the critical section while holding the lock.

In a shared environment where multiple tenants can compete to acquire the shared synchronization primitives while trying to access the

shared data structures that are part of the shared infrastructure, we have shown that an unfair lock usage can lead to two problems. While scheduler subversion problem deals with poor performance, adversarial synchronization deals with poor performance and denial-of-service.

Scheduler subversion is an imbalance in the CPU allocation that arises when the lock usage pattern dictates the CPU allocation instead of the CPU scheduler. When scheduler subversion occurs, the thread that dwells longer in the critical section becomes the dominant user of the CPU. Using a real-world application, we demonstrated the problem of scheduler subversion. There are two reasons why scheduling subversion occurs for applications that access locks. First, scheduler subversion may occur when the critical sections are of significantly different lengths. Second, scheduler subversion may occur when threads spend the majority of their runtime in the critical section.

To explain the idea of adversarial synchronization, we introduced a new class of attacks – synchronization and framing attacks that can exploit synchronization primitives to harm applications' performance in a shared environment. An unprivileged malicious actor can control the duration of the critical sections to stall victims trying to access the shared synchronization primitives. Furthermore, in framing attacks, even after the malicious actor quiesces, the victims observed poor performance as they continue to access the expanded data structure.

We illustrated three different attacks on the Linux kernel data structures using containers and showed how a malicious actor could employ different methods to launch synchronization and framing attacks.

Thus, in the first part of the dissertation, we showed how lock usage is an important property. We showed how an imbalance in the lock usage could lead to two problems – performance and security. While in a cooperative environment, the effects of unfair lock usage can be ignored as the effect is only seen within the application, for shared environments,

where multiple tenants access the shared infrastructure; it becomes crucial to avoid and handle unfair lock usage.

7.1.2 Scheduler-Cooperative Locks

In the second part of the dissertation, we focused on the scheduler subversion problem and defined the idea of lock usage fairness. Lock usage fairness guarantees that each competing entity receives a window of opportunity to use the lock one or more times.

We started by studying lock usage fairness and understanding how existing locks can lead to scheduler subversion. Based on that study, we showed that the non-preemptive nature of the locks makes it difficult for schedulers to allocate resources when each thread may hold a lock for a different amount of time.

We then introduced a new metric – lock opportunity to measure lock usage fairness. Lock opportunity is defined as the amount of time a thread holds a lock or could acquire the lock when the lock is free. Using the idea of lock opportunity, we then introduced Scheduler-Cooperative Locks (SCLs), which can track lock usage and adjust the lock opportunity time to ensure proportional lock usage fairness and align with the CPU scheduling goals.

There are three crucial components to design SCLs – tracking and accounting for the lock usage, penalizing the threads depending on the lock usage, and providing dedicated lock opportunity using lock slice. Using these three components, we discussed the design and implementation of three different types of SCLs – a user-level mutex lock (u-SCL), a reader-writer lock (RW-SCL), and a simple kernel implementation (k-SCL). While u-SCL and k-SCL guarantee lock usage fairness at a thread level, RW-SCL guarantees lock usage fairness based on the thread's work (i.e., readers and writers).

Using microbenchmarks, we showed that SCLs are efficient, scale well, and can achieve high performance with minimal overhead under extreme workloads while still guaranteeing lock usage fairness under a variety of synthetic lock usage scenarios.

Lastly, to show the effectiveness of SCLs, we ported the SCLs in two user-space applications – UpScaleDB and KyotoCabinet, and the Linux kernel. In all the three cases, regardless of the lock usage patterns, SCLs ensured that each thread receives proportional lock allocations that match those of the CPU schedulers.

7.1.3 Taming Adversarial Synchronization Attacks using Trāṭṛ

In the last part of the dissertation, we switched the focus to the security issues arising from unfair lock usage. First, we discussed how existing techniques used to address algorithmic complexity attacks such as universal hashing, randomized algorithms, partitioning, etc., are insufficient to address the synchronization and framing attacks. Even though containers can guarantee better isolation, the existing isolation techniques are inadequate to ensure strong performance isolation.

Based on the experience with the synchronization and framing attacks, we designed and implemented Trāṭṛ, a Linux extension, to defend against the synchronization and framing attacks. Unlike existing solutions, Trāṭṛ provides a common framework to tackle the problem of adversarial synchronization.

Trāṭṛ comprised of four mechanisms – tracking, detecting, preventing, and recovery. The tracking mechanism tracks the kernel objects allocation per user and how each user contributes to the data structure size. The detection mechanism periodically probes the synchronization primitives associated with vulnerable data structures and monitors the synchroniza-

tion stalls. If the stalls are longer than a particular threshold, an attack is detected, and Trātṛ uses the tracking information to identify the attacker.

The last two mechanisms dealt with the mitigation of the attack. On determining the attacker, the prevention mechanism kicks in to prevent the attack from worsening by stopping the attacker from allocating more objects. The recovery mechanism then cleans up the data structure to pre-attack state so that the performance can be restored to baseline performance.

Using microbenchmarks, a benchmarking suite, and real-world applications, we showed that Trātṛ can quickly detect an attack and effectively prevent the attack from worsening and perform recovery instantaneously. Furthermore, the performance impact on the victim in the presence of Trātṛ is minimal, showing the coordination of the four mechanisms. We also performed an extensive overhead study and showed that there is only 0-4% tracking overhead incurred while there is less than 1.5% impact on the performance due to the other three mechanisms in the absence of an attack. Lastly, we showed how Trātṛ can detect simultaneous attacks without impacting the victims' performance.

7.2 Lessons Learned

In this section, we present a list of general lessons we learned while working on this dissertation.

Existing "strongest" guarantees are not the strongest. Lock acquisition fairness is considered the strongest variant to guarantee that any given thread can never be prevented from making forward progress. By bounding the wait times to acquire the locks, each thread will be guaranteed forward progress. While bounding the wait times, the assumption is that the critical sections will be similarly sized, and hence, all the waiting times to acquire the lock for all the threads will be similar.

However, in this dissertation, we showed that critical section sizes are not similar. As the data structure grows and shrinks, the critical section sizes vary significantly. In a shared environment with varying critical section sizes, applications will face performance and security problems such as scheduler subversion or adversarial synchronization. When one thread starts dominating the lock usage, the wait times of the victims will differ by six to seven orders of magnitude, and it continues to increase as the attack continues.

We view such a higher order of magnitude difference in the wait times as not the strongest variant to guarantee that any given thread can never be prevented from progressing. Therefore, a better, stronger variant is needed to keep the wait times bounded even with varying critical sections. This dissertation takes the initial but essential step in this direction by considering lock usage fairness as a critical lock property.

For more robust performance isolation, data structures and their synchronization primitives need to be considered as a resource. So far, the researchers and practitioners believed that by controlling the usage of the four primary resources – the CPU, the memory, the disk, and the network, one could guarantee strong performance isolation [47, 72, 81, 100, 112, 143, 158]. There has been a plethora of research work that continues to improve tenant-wide performance isolation and fairness guarantees.

Through this dissertation, we showed that in a shared environment, the high degree of sharing across tenants through the shared infrastructure creates an avenue for performance interference. When the tenants access the shared data structures and synchronization primitives, due to the variation in the workloads and the data structures state tenants can observe poor performance and denial-of-service.

This dissertation suggests that the data structures and the associated synchronization primitives should be considered resources along with the four primary resources. Furthermore, fair usage of the data struc-

tures and the synchronization primitives must be considered to guarantee strong performance isolation. Thus, this work is the first step towards more stronger performance isolation.

Linux Kernel's attack surface is wide and needs a common approach to handling adversarial synchronization. In this dissertation, we showed three different attacks on three different data structures by using different methods. As the Linux kernel comprises hundreds of data structures, the attack surface of the Linux kernel is wide. Therefore, there can be many different ways to exploit the data structures.

Adversarial synchronization is similar to algorithmic complexity attacks. Like algorithmic complexity attacks, there can be numerous solutions to address adversarial synchronization. Or rather, it is possible to opt for solutions that address algorithmic complexity attacks to avoid adversarial synchronization. However, as discussed earlier, all the solutions cannot handle the large attack surface and the different ways an attack can be launched.

Therefore, there is a need to have a common solution to address adversarial synchronization. We build a common solution through this thesis by understanding the key aspects of the attacks and formalizing the attacks. The solution can be easily expanded to include more vulnerable data structures whenever a new attack or a vulnerability is identified through various means.

7.3 Future Work

We now outline directions in which this work could be extended in the future.

7.3.1 Expand SCLs to Support Other Schedulers and Locks

In this dissertation, while designing the Scheduler-Cooperative Locks, the focus has been on the Linux kernel's CFS scheduler only. However, for other operating systems such as FreeBSD that use the ULE scheduler, we need to understand the impact of lock slice sizes. Moreover, other schedulers might observe scheduler subversion too. Therefore, we find it imperative to study the impact of the scheduler subversion problem and SCLs with other available schedulers.

To support a variety of scheduling goals, the implementation of SCLs needs to be accordingly adjusted. We would like to extend SCLs to support other schedulers, such as priority scheduling. The SCLs should contain multiple queues to support priorities and grant lock access depending on the priority to support priority scheduling.

The design of SCLs comprises three components – tracking and accounting lock usage, penalizing threads depending on the lock usage, and providing dedicated lock opportunity using lock slice. Using these three components, we showed how existing locks (K42 and reader-writer locks) could be converted to guarantee lock usage and lock acquisition fairness across different schedulable entities. Similarly, conducting another exercise to convert other existing locks to support lock usage fairness while preserving their other existing properties is possible.

7.3.2 SCLs in Multiple Locks Situation

So far, the focus in this dissertation has been around using SCLs within a single lock situation. Currently, when a thread has used up its lock usage quota, SCLs will force the threads to sleep so that other threads can get an opportunity to acquire the lock.

However, penalizing threads in a hierarchical locking scenario may

not work well as the thread may be holding on to a higher level lock while it is penalized for a lower level lock. In doing so, the critical section of the higher-level lock will increase, leading to performance degradation. More so, it is entirely possible that after acquiring the higher-level lock, preemption may be disabled, leading to problems when the lower level SCL forces a thread to sleep.

There can be similar other situations arising where the fairness goals of one of the lock may interfere with the fairness goals of another lock leading to performance degradation. As most complex concurrent systems use multiple locks, it will be interesting to understand the interaction of multiple SCLs in such a setting. In particular, the use of machine learning to learn the lock usage patterns may be useful in a hierarchical locking scenario so that all the SCLs can coordinate amongst themselves to ensure that no thread is penalized while holding higher-level locks.

7.3.3 Work Conserving Nature of SCLs

One of the crucial components to ensure lock usage fairness is to provide a dedicated window of opportunity to all the threads using lock slice. However, in doing so, SCLs are non-work-conserving relative to locks. There can be scenarios where a thread will not acquire the lock again during a lock slice, keeping the lock-free even though other threads are waiting to acquire the lock.

To ensure that the window of opportunity is used effectively by all the threads, we believe there are a couple of options. The first option is to observe the lock usage patterns and club one or more threads into a single lock slice. In doing so, these threads will take turns to acquire the lock within a lock slice. Even if one thread may not acquire the lock immediately, other threads may acquire the lock shortly, thereby not wasting the opportunity granted to all these threads. One of the goals should be to ensure that two threads having diverging lock usage patterns should be

clubbed to complement and cooperate within the lock slice.

The second option is to provide additional APIs such that the threads can give advice or directions to the SCLs about its lock usage behavior. This idea is similar to the madvise idea where the threads can inform the kernel about address range to improve the performance [87]. For example, if a thread is not going to utilize the entire lock slice, it can inform the SCL about it while unlocking so that the lock slice may be prematurely transferred to other waiting threads.

7.3.4 Scheduler-driven fairness

So far, we have discussed the design of SCLs that can guarantee lock usage fairness in various lock usage scenarios. Additionally, we showed how SCLs can allocate the lock usage that aligns with the CPU scheduling goals, thereby avoiding the scheduler subversion problem. However, we view that SCLs are one way of solving the scheduling subversion problem by designing a lock that guarantees lock usage fairness. We believe that another alternative way of solving the subversion problem involves the CPU scheduler itself.

Locks are an extension of the CPU scheduler as they also schedule the threads depending on which thread will acquire the lock next and which threads will wait for their turn. Depending on the type of the lock, either the threads spin or block to acquire the lock. The CPU scheduler can arbitrarily schedule threads that are not in sync with the locking strategy as the scheduler is unaware of the locks and how the threads access the locks.

There are other performance-related problems associated with the locks and scheduler, such as the Lock Holder Preemption problem [154] and Lock Waiter problem [151]. Several solutions have been proposed to avoid the LHP, and LWP [12, 83, 118, 151, 152, 161]. However, all these

solutions tolerate the LHP and LWP by limiting the side effects of the LHP and LWP and degrade overall performance.

To mitigate the fairness and performance problems, we believe that scheduler-driven fairness can be an interesting avenue for future work where the scheduler treats locks as a resource and effectively allocates the resource to all the threads fairly. As the locks invoke the scheduler when acquiring and releasing the lock, the scheduler can track the lock usage pattern of each thread for all the locks in the system and penalize the threads by not scheduling the threads. Moreover, the locks do not have to support various scheduling algorithms. The scheduler can take care of adjusting the penalty depending on the scheduling algorithm.

Scheduler-driven fairness can also be used to address LHP and LWP problems. To avoid LHP, the scheduler will not be allowed to preempt the thread until the lock is released by extending the time slice of the running thread. To avoid LWP, the scheduler will keep track of the next thread and ensure that it gets immediately scheduled after the current owner releases the lock.

We anticipate several challenges while implementing scheduler-driven fairness. First, the scheduler has to be aware of all the locks within the system. Getting all the locks information must be either done at compilation time or dynamically at run time. Sometimes, it is impossible to get all the information at compile time as many locks might be initialized dynamically. So the desired approach needs to incorporate both methods to capture all the information.

Another problem we expect is the interaction between the kernel and userspace. The locks might be acquired in user space while the scheduler runs in kernel space. One will have to devise a way to exchange the information between the kernel and userspace.

The scheduler is supposed to be lightweight. With all the information needed to guarantee fairness, the scheduler will be over-burdened,

and we need to design new data structures to let the scheduler seek all information without slowing down.

There can also be problems concerning hierarchical locking or nested locking. For example, consider a situation where the thread has acquired the outer lock but cannot acquire the inner lock due to some reason. As the thread cannot be preempted as it holds the outer lock, there can still be problems like LHP or LWP.

The scheduler-driven fairness approach offers multiple advantages over the lock-driven fairness approach. The scheduler monitors the lock usage independent of the lock type, and thus any type of lock can be accommodated to support lock usage fairness. Additionally, one does not have to worry about supporting different scheduling policies as we do for SCLs. Lock-based fairness needs to constantly update the status of the threads and scheduling goals. With scheduler-driven fairness, the scheduler is already aware of the threads' status and priorities; it will be easier to adapt to the dynamic nature of the workloads.

7.3.5 Analyzing Linux Kernel to find Vulnerable Data Structures

The Linux kernel comprises hundreds of kernel data structures that are being protected by thousands of synchronization primitives. In this dissertation, we showed the problems on three data structures only. However, the problem may be widespread, given that there are numerous instances of synchronization primitives.

An attacker can always target the critical sections where there is a possibility to elongate the critical section size. For example, if a critical section contains a loop whose termination criteria depends on the data structure's state, the attacker can exploit this fact to build the data structure state such that the loop is executed for thousands or tens of thousands of iterations, making the critical section size longer.

We find it imperative to study the critical sections protected by these thousands of synchronization primitives. Furthermore, it is crucial to understand all the properties of the critical sections to identify how an attacker can exploit certain critical section properties.

We believe that through static analysis of the Linux kernel source code, one can identify vulnerable data structures and the associated critical sections that can be targeted. However, it will be very hard to identify vulnerable data structures at runtime as not all code paths get executed while running workloads. It will need a humongous effort to identify and execute all the workloads that will cover all the code paths.

However, static analysis on the Linux kernel's source code will not be easy as there exist varied coding patterns, debug code, etc. We anticipate many challenges, such as identifying function pointers, etc., through static analysis. But, the reward of such an exercise is enormous. One can use the critical section and synchronization primitives analysis to identify potential vulnerable data structures that an attacker can target. The same analysis can identify the bottlenecks to guarantee strong performance isolation in the Linux kernel. Therefore, we believe it will be an interesting avenue of future work to analyze the Linux kernel.

7.3.6 Combining SCLs and Trātṛ

Currently, Trātṛ relies on probing to identify if a synchronization primitive is being under an attack. If the synchronization stalls while probing exceed the set threshold values, Trātṛ flags an attack. Once an attack is detected, Trātṛ has to walk through the data structure to identify the attacker. However, it is possible that a victim may be falsely flagged as an attacker and may be penalized for no reason.

On the other hand, SCLs have to track the lock usage and account for all the lock usage so that it can correctly identify who the dominant users are and then accordingly penalize such dominant users. Other than SCLs,

no other locks track such usage and hence Trātṛ has to rely on probing and traversing the data structure to detect an attack and identify an attacker.

For both the synchronization and framing attacks, SCLs will treat either the attacker or the victims holding the synchronization primitives for a longer duration as dominant users. We strongly believe that combining SCLs and Trātṛ, Trātṛ can use the information that SCLs collect about the lock usage to detect an attack. Once Trātṛ knows that an attack is underway, it can traverse the data structure to identify who is responsible for expanding the data structure and appropriately flag that user as an attacker. By doing so, Trātṛ can avoid probing the synchronization primitives and reduce the impact of participating in the lock acquisition process.

We feel there is another advantage of combining SCLs and Trātṛ. As seen in Section 5.3.7, a defense-aware attacker can easily remain undetected by limiting the critical section sizes below the threshold limits. If SCLs are used instead of the existing locks, in such situations, SCLs will be able to identify that the attacker is a dominant user and accordingly penalize the attacker. Therefore, the interference created by a defense-aware attacker will be further diminished, helping the victims. We believe that Trātṛ can easily help convert a synchronization or framing attack into a minor performance inconvenience. With SCLs, the minor performance inconvenience can be further reduced, leading to stronger performance isolation guarantees.

7.3.7 Attacks on Concurrency Control Mechanisms

In this dissertation, we only show how synchronization primitives can be targeted to launch attacks leading to poor performance and denial-of-services. Locks and other mutual exclusion primitives are a form of pessimistic concurrency control mechanism.

However, it is entirely possible that other forms of concurrency control

mechanisms can be targeted too. For example, the optimistic concurrency control mechanism allows to continue with the operation until its end without blocking the operations and then aborting the operation if any other operation interferes. Upon aborting the operation, the operation is restarted, hoping that no other operation will interfere.

We believe that there is scope for an attack with optimistic concurrency control mechanisms where an attacker continuously interferes with other concurrent operations by always committing the operation with minimal changes. It will be interesting to study the adversarial aspects of optimistic concurrency control in a shared environment.

7.3.8 Opportunity-based fairness for other non-preemptive resources

We have shown that solutions proposed for preemptive resources such as CPU, disk, memory, and network cannot apply to non-preemptive resources like locks. In this dissertation, we show how opportunity-based fairness can be used to guarantee lock usage fairness. Other non-preemptive resources such as GPU can also benefit by applying the idea of opportunity-based fairness.

Currently, GPU commands are non-preemptive and hence subject to priority inversions. Moreover, it is hard to perform schedulability analysis due to the non-preemptive nature [24]. Like locks, GPUs can also be subjected to unfair usage in a shared environment, leading to performance and security problems. Thus, we believe that it is possible to extend the idea of opportunity-based fairness to other non-preemptive resources and guarantee that such resources are shared fairly in a shared environment.

7.4 Closing Words

Locks and other synchronization primitives that ensure atomicity and ordering guarantees in concurrent systems have been considered cooperative so far. This assumption worked so far in a cooperative environment where the effects and the impact of synchronization can be largely ignored or refactored in solely by the application developers.

With the rapid adoption of shared environments where multiple tenants execute their programs or requests on shared infrastructure, the role of synchronization primitives need to be viewed differently. In such environments, locks and other synchronization primitives should be viewed from a competitive perspective so that every tenant gets an equal opportunity to access the shared synchronization primitives. This equal opportunity can be guaranteed by ensuring lock usage fairness.

Lock usage fairness is an extension of the existing fairness property to provide even stronger guarantees where all the threads can make forward progress with or without the presence of malicious actors. By viewing locks as competitive and guaranteeing fair lock usage, one can move closer to more stronger performance isolation. This dissertation is a step towards guaranteeing a stronger variant of fairness and ensuring strong performance isolation.

Bibliography

[1] CVE-2003-0718. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0718.

[2] CVE-2004-0930. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0930.

[3] CVE-2005-0256. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0256.

[4] CVE-2005-1807. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1807.

[5] CVE-2005-2316. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2316.

[6] CVE-2007-1285. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1285.

[7] CVE-2008-2930. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2930.

[8] CVE-2008-3281. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3281.

[9] CVE-2011-1755. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1755.

[10] Jose L Abell, Juan Fern, Manuel E Acacio, et al. Glocks: Efficient support for highly-contended locks in many-core cmps. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 893–905. IEEE, 2011.

[11] Nitin Agrawal, Leo Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Emulating goliath storage systems with david. *ACM Transactions on Storage (TOS)*, 7(4):1–21, 2012.

[12] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk 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*, MICRO-47, pages 394–405, 2014.

[13] Tigran Aivazian. Inode Caches and Interaction with Dcache. `https://tldp.org/LDP/lki/lki-3.html`.

[14] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313, 2017.

[15] Inc. Amazon Web Services. Firecracker. `https://firecracker-microvm.github.io/`.

[16] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[17] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: A study of firecracker and gvisor. In

Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, page 101–113, 2020.

[18] Jelena Antić, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. Locking made easy. In *Proceedings of the 17th International Middleware Conference*, page 20, 2016.

[19] AppArmor. AppArmor Documentation. `https://gitlab.com/apparmor/apparmor/-/wikis/Documentation`.

[20] Jonathan Appavoo, Marc Auslander, Maria Butrico, Dilma M Da Silva, Orran Krieger, Mark F Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W Wisniewski, and Jimi Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[21] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[22] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 45–58.

[23] Noa Bar-Yosef and Avishai Wool. Remote algorithmic complexity attacks against randomized hash tables. In *International Conference on E-Business and Telecommunications*, pages 162–174. Springer, 2007.

[24] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in gpgpus. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 287–296, 2012.

[25] Andrew D Birrell. Implementing condition variables with semaphores. In *Computer Systems*, pages 29–37. Springer, 2004.

[26] Leonid Boguslavsky, Karim Harzallah, A Kreinen, K Sevcik, and Alexander Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 21(2):246–254, 1994.

[27] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena. USENIX ATC '18, 2018.

[28] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *OSDI '08*, December 2008.

[29] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. OSDI'10, page 1–16, 2010.

[30] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[31] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. USENIX ATC'10.

[32] Björn B. Brandenburg and James H. Anderson. Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems. *Real-Time Systems*, 46:25–87, 2010.

[33] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 27–41, 2009.

[34] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *SIGPLAN*, pages 157–166, 2013.

[35] Remy Card. tune2fs(8) - linux man page. `https://linux.die.net/man/8/tune2fs`, 2018.

[36] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems. In *27th International Symposium on Parallel and Distributed Processing*, pages 1309–1320, 2013.

[37] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level numa systems. PPoPP 2015, pages 215–226, 2015.

[38] Milind Chabbi and John Mellor-Crummey. Contention-conscious, locality-preserving locks. *SIGPLAN Not.*, 51(8):22:1–22:14, 2016.

[39] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and Jose F Martinez. Workload Characterization of Interactive Cloud Services on Big and Small Server Platforms. In *IISWC*, pages 125–134, 2017.

[40] Chris Rivera and Robert Love. slabtop(1) - Linux manual page. `https://man7.org/linux/man-pages/man1/slabtop.1.html`.

[41] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic data structures in adversarial environments. In *SIGSAC*, pages 1317–1334, 2019.

[42] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[43] Jonathan Corbet. How to get rid of mmap_sem. `https://lwn.net/Articles/787629/`.

[44] Jonathan Corbet. SipHash in the kernel. `https://lwn.net/Articles/711167/`.

[45] TravisS. Craig. Building fifo and priority-queuing spin locks from atomic swap. Technical report, 1993.

[46] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*, pages 29–44, 2003.

[47] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 7(1):37–48, September 2013.

[48] Dice David. Brief announcement: a partitioned ticket lock.. 309-310., 2011.

[49] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. *ACM SIGPLAN Notices*, 49(10):291–307, 2014.

[50] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. SOSP '13, pages 33–48.

[51] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *USENIX ATC'19*, pages 693–708.

[52] Mathieu Desnoyers, Michel R. Dagenais, Jonathan Walpole, Paul E. McKenney, and Alan S. Stern. User-Level Implementations of Read-Copy Update. IEEE Transactions on Parallel & Distributed Systems, February 2012.

[53] Dave Dice. Malthusian locks. *CoRR*, abs/1511.06035, 2015.

[54] Dave Dice and Alex Kogan. BRAVO: Biased Locking for Reader-Writer Locks. USENIX ATC '19, page 315–328.

[55] Dave Dice and Alex Kogan. Compact NUMA-Aware Locks. EuroSys '19.

[56] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. SPAA '11, pages 65–74.

[57] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[58] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Radmin: early detection of application-level resource exhaustion and starvation attacks. In *International Symposium on Recent Advances in Intrusion Detection*, pages 515–537. Springer, 2015.

[59] D.H.J. Epema. An analysis of decay-usage scheduling in multiprocessors. SIGMETRICS '95, 1995.

[60] Fabio Kung, Sargun Dhillon, Andrew Spyker, Kyle Anderson, Rob Gulewich, Nabil Schear, Andrew Leung, Daniel Muinom and Manas Alekar. Evolving Container Security with Linux User Namespaces. `https://netflixtechblog.com/evolving-container-security-with-linux-user-namespaces-afbe3308c082`.

[61] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking energy. USENIX ATC '16, pages 393–406.

[62] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. *SIGPLAN Not.*, 47(8):257–266, February 2012.

[63] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, volume 85, 2002.

[64] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI' 99*, pages 87–100.

[65] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[66] Amir Reza Ghods. A study of linux perf and slab allocation subsystems. Master's thesis, University of Waterloo, 2016.

[67] Gianluca Borello. Container isolation gone wrong. `https://sysdig.com/blog/container-isolation-gone-wrong/`.

[68] Marc Girault, Robert Cohen, and Mireille Campana. A generalized birthday attack. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 129–156. Springer, 1988.

[69] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. *SIGPLAN Not.*, 50(8):1–10, January 2015.

[70] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.*, 36(1), March 2019.

[71] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore locks: The case is not closed yet. In *USENIX ATC '16*, pages 649–662.

[72] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. Middleware '06, pages 342–362.

[73] Bijun He, William N. Scherer, and Michael L. Scott. Preemption adaptivity in time-published queue-based spin locks. HiPC'05, pages 7–18.

[74] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA '10, pages 355–364.

[75] Maurice Herlihy. Wait-Free Synchronization. *Transactions on Programming Languages*, 11(1), January 1991.

[76] Wen-mei Hwu, Kurt Keutzer, and Timothy G Mattson. The concurrency challenge. *IEEE Design & Test of Computers*, 25(4):312–320, 2008.

[77] Docker Inc. Docker Engine managed plugin system. `https://docs.docker.com/engine/extend/`.

[78] Docker Inc. Vulnerability scanning for Docker local images. `https://docs.docker.com/engine/scan/`.

[79] Google Inc. gVisor. `https://gvisor.dev/`.

[80] Raj Jain, Dah-Ming Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998.

[81] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. Eyeq: Practical network performance isolation for the multi-tenant cloud. HotCloud'12.

[82] F Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C Mowry. Decoupling contention management from scheduling. ASPLOS XV, pages 117–128, 2010.

[83] J. Fitzharding K. Raghavendra. Paravirtualized ticket spinlocks . `https://lwn.net/Articles/552696/`.

[84] Anna R Karlin, Kai Li, Mark S Manasse, and Susan Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. *ACM SIGOPS Operating Systems Review*, 25(5):41–55, 1991.

[85] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. SOSP '19, pages 586–599.

[86] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable numa-aware blocking synchronization primitives. USENIX ATC '17, pages 603–615.

[87] Michael Kerrisk. madvise(2) — Linux manual page. `https://man7.org/linux/man-pages/man2/madvise.2.html`.

[88] Michael Kerrisk. pthread_spin_lock. `https://man7.org/linux/man-pages/man3/pthread_spin_lock.3.html`.

[89] Suraiya Khan and Issa Traore. A prevention model for algorithmic complexity attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 160–173. Springer, 2005.

[90] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. ICPP '93, pages 201–204.

[91] Kaz Kylheku. What is PTHREAD_MUTEX_ADAPTIVE_NP? `https://stackoverflow.com/questions/19863734/what-is-pthread-mutex-adaptive-np`.

[92] Lawrence Livermore National Laboratory. Mutex variables. `https://computing.llnl.gov/tutorials/pthreads`, 2017.

[93] FAL Labs. KyotoCabinet. `https://fallabs.com/kyotocabinet/`.

[94] Michael Larabel. FUSE Gets User Namespace Support With Linux 4.18. `https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.18-FUSE`.

[95] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable Reader-Writer Locks. SPAA '09, page 101–110.

[96] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. *ACM SIGOPS Operating Systems Review*, 28(5):25–35, 1994.

[97] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. ACSAC '18, pages 418–429.

[98] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-Section

Execution to Improve the Performance of Multithreaded Applications. USENIX ATC'12.

[99] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Fast and portable locking for multicore architectures. *ACM Trans. Comput. Syst.*, 33(4):13:1–13:62, 2016.

[100] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. OSDI'14, pages 81–96.

[101] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical clh queue lock. Euro-Par'06, pages 801–810.

[102] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. ISPP '94, pages 165–171.

[103] A Martin, S Raponi, T Combe, and R Di Pietro. Docker ecosystem–vulnerability analysis. *Computer Communications*, 122:30–43, June 2018.

[104] Paul E. McKenney. The new visibility of RCU processing. `https://lwn.net/Articles/518953/`.

[105] Paul E. McKenney. What is RCU, Fundamentally? `https://lwn.net/Articles/262464/`.

[106] Paul E. McKenney. *Selecting Locking Designs for Parallel Programs*, page 501–531. 1996.

[107] Paul E McKenney. Is parallel programming hard, and, if so, what can you do about it? *arXiv preprint arXiv:1701.00854*, 2017.

[108] Paul E. McKenney and Jonathan Walpole. What is rcu, fundamentally? *Linux Weekly News (LWN. net)*, 2007.

[109] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[110] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. *SIGPLAN Not.*, 26(7):106–113, April 1991.

[111] John M. Mellor-Crummey and Michael L. Scott. Synchronization without Contention. *SIGARCH Comput. Archit. News*, 19(2):269–278, April 1991.

[112] Paul Menage. CGroup documentation. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`, 2018.

[113] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[114] Michael Kerrisk. Namespaces in operation, part 5: User namespaces. `https://lwn.net/Articles/772885/`.

[115] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding Manycore Scalability of File Systems. USENIX ATC '16, page 71–85.

[116] Neil Brown. Linux kernel design patterns - part 2. `https://lwn.net/Articles/336255/`.

[117] M. Oh, J. Eom, J. Yoon, J. Y. Yun, S. Kim, and H. Y. Yeom. Performance Optimization for All Flash Scale-Out Storage. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 316–325, Sep. 2016.

[118] Jiannan Ouyang and John R. Lange. Preemptable ticket spinlocks: Improving consolidated performance in the cloud. *SIGPLAN Not.*, 48(7):191–200, March 2013.

[119] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. PDSIA '99.

[120] Sujin Park, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Contextual concurrency control. HotOS '21, pages 167–174.

[121] Yuvraj Patel, Ye Chenhao, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Using Trāṭr to tame adversarial synchronization. In Submission.

[122] Yuvraj Patel, Mohit Verma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Revisiting concurrency in high-performance nosql databases. HotStorage '18.

[123] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding scheduler subversion using scheduler-cooperative locks. EuroSys '20.

[124] David Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 2010.

[125] Ben Pfaff. Performance analysis of bsts in system software. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):410–411, 2004.

[126] Phoronix Media. Phoronix Test Suite - Linux Testing and Benchmarking Platform, Automated Testing, Open-Source Benchmarking. `https://www.phoronix-test-suite.com`.

[127] Aravinda Prasad and K. Gopinath. Prudent memory reclamation in procrastination-based synchronization. ASPLOS '16, page 99–112.

[128] Aravinda Prasad, K. Gopinath, and Paul E. McKenney. The rcu-reader preemption problem in vms. USENIX ATC '17, pages 265–270.

[129] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xmp: Selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy*, pages 563–577, 2020.

[130] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *ACM SIGOPS Operating Systems Review*, 36(SI):45–60, 2002.

[131] Zoran Radovic and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. HPCA '03.

[132] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Transactions on networking*, 17(1):26–39, 2008.

[133] David P Reed and Rajendra K Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, 1979.

[134] Jeff Roberson. ULE: A Modern Scheduler for FreeBSD. In *BSDCon*, 2003.

[135] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. SOSP '17, pages 342–358.

[136] Dipankar Sarma and Paul E. McKenney. Making rcu safe for deep sub-millisecond response realtime applications. USENIX ATC '04, pages 182–191.

[137] Bryan Schauer. Multicore processors–a necessity. *ProQuest discovery guides*, pages 1–14, 2008.

[138] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[139] Michael L. Scott and William N. Scherer. Scalable queue-based spin locks with timeout. PPoPP '01, pages 44–52.

[140] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: an Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep. 1990.

[141] Jianchen Shan, Xiaoning Ding, and Narain Gehani. Apples: Efficiently handling spin-lock synchronization on virtualized platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1811–1824, 2016.

[142] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. CODASPY '17, page 269–280.

[143] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. OSDI'12, pages 349–362.

[144] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. USENIX Security 18, pages 361–376.

[145] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.

[146] Herb Sutter and James Larus. Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry. *Queue*, 3(7):54–62, 2005.

[147] Sriram Swaminathan, J. Stultz, J. F. Vogel, and Paul McKenney. Abstract fairlocks- a high performance fair locking scheme. 2003.

[148] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. SIGPLAN '10, pages 269–280.

[149] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[150] Gadi Taubenfeld. Fair synchronization. *J. Parallel Distrib. Comput.*, 97:1–10, 2016.

[151] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). EuroSys '17, pages 286–297.

[152] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. EuroSys '16, pages 3:1–3:14.

[153] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E Porter. How to get more value from your file system directory cache. SOSP '15, pages 441–456.

[154] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. VM'04.

[155] UpScaleDB Inc. UpScaleDB. `https://upscaledb.com/`.

[156] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To fuse or not to fuse: Performance of user-space file systems. FAST '17, pages 59–72.

[157] Axel Wagner. ext4: Mysterious "No space left on device"-errors. `https://blog.merovius.de/2013/10/20/ext4-mysterious-no-space-left-on.html`, 2018.

[158] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. OSDI '02.

[159] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. OSDI '94.

[160] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO diaries: Application-controlled frequency scaling explained. USENIX ATC '14.

[161] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. PACT '06, pages 124–133.

[162] David Wentzlaff and Anant Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.

[163] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant

Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. SoCC '10, page 3–14.

[164] Mingzhe Zhang, Haibo Chen, Luwei Cheng, Francis CM Lau, and Cho-Li Wang. Scalable adaptive numa-aware lock. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1754–1769, 2016.

[165] A. Zhong, H. Jin, S. Wu, X. Shi, and W. Gen. Optimizing Xen Hypervisor by Using Lock-Aware Scheduling. In *2012 Second International Conference on Cloud and Green Computing*, pages 31–38, Nov 2012.