EPFL

Thèse n° 7141

Distributed Computing with Modern Shared Memory

Présentée le 20 novembre 2020

à la Faculté informatique et communications
Laboratoire de calcul distribué
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Mihail Igor ZABLOTCHI

Acceptée sur proposition du jury

Prof. R. Urbanke, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. H. Attiya, rapporteuse
Prof. N. Shavit, rapporteur
Prof. J.-Y. Le Boudec, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

"Where we going man?"
"I don't know but we gotta go."
— Jack Kerouac, On the Road

To my grandmother Momo,
who got me started on my academic adventure
and taught me that learning can be fun.

Acknowledgements

I am first and foremost grateful to my advisor, Rachid Guerraoui. Thank you for believing in me, for encouraging me to trust my own instincts, and for opening so many doors for me. Looking back, it seems like you always made sure I had the perfect conditions to do my best possible work. Thank you also for all the things you taught me: how to get at the heart of a problem, how to stay optimistic despite all signs to the contrary, how to be pragmatic while also chasing beautiful problems.

I am grateful to my other supervisors and mentors: Yvonne Anne Pignolet and Ettore Ferranti for my internship at ABB Research, Maurice Herlihy for the time at Brown University and at Oracle Labs, Dahlia Malkhi and Ittai Abraham for my internship at VMware, Virendra Marathe and Alex Kogan for my internship at Oracle Labs and the subsequent collaboration, Marcos Aguilera for the collaboration on the RDMA work, and Aleksandar Dragojevic for the internship at Microsoft Research. Thank you for spoiling me with such interesting problems to work on and for teaching me so many valuable lessons about research.

I am grateful to the members of my PhD oral exam jury: Prof. Hagit Attiya, Prof. Jean-Yves Le Boudec, and Prof. Nir Shavit. Thank you for your insightful questions and comments, which have helped improve my thesis. I am also thankful to Prof. Rüdiger Urbanke for presiding over my thesis committee.

I am grateful to my collaborators: Naama Ben-David, Thanasis Xygkis, Karolos Antoniadis, Vincent Gramoli, Vasilis Trigonakis, Tudor David, Oana Bălmău, Julien Stainer, Peva Blanchard, and Nachshon Cohen. Discussing and working with you on tough problems has undoubtedly been the most enjoyable part of my PhD. Huge thanks to Naama and Thanasis for the hundreds of hours of peer programming, Skype meetings and deadline crunches that we shared in the last two years.

I am grateful to the administrative staff of the lab. Thank you, France, for your invaluable help on so many fronts, big and small. I always look forward to interacting with you because I know you will help me get things done quickly, and your green office feels like an oasis where I can take a breath of fresh air. I also always enjoy our chats on many different subjects. Thank you, Fabien, for your help with the technical infrastructure for my experiments; thanks to you, it has always been smooth sailing in that regard.

I am grateful to my colleagues and friends in the Distibuted Computing Lab at EPFL, who have made it fun to come into work every day: Karolos, Matej, George D., Adi, George C., Thanasis, Sébastien, Le, Mahdi, Alex, Jovan, Arsany, Vlad, Tudor, David, Vasilis, Jingjing, Mahsa, Matteo, Rhicheek, Victor, and Andrei. Thanks for all the goofy memories that have helped me make light of the often stressful PhD life. Special thanks to Karolos for always being extra supportive in my moments of doubt, and for the countless hours of (sometimes scientific) discussions. Big thanks to Le for helping with the French translation of the thesis abstract.

I am grateful to my friends outside of the lab: Corina, Camilo, Cristina, Bogdan Stoica, Olivier, Maria, Viki, Étienne, Laura, Bogdan Sassu, Călin, Răzvan, and Diana. Thank you for the conversations, laughs, hikes, parties, and adventures that we shared. Without you the PhD would have been a much lonelier experience. I am also thankful to all the interesting people I met during my internships: Irina, Dan, Aran, Sasha, Heidi, Mike, Zoe, Rashika, Lily, Petros, Dario, and Ho-Cheung. Thank you for the fun memories!

I am deeply grateful to my girlfriend, Monica, who has been by my side for most of the PhD. Thank you for supporting and encouraging me during my most difficult moments. You have been the best adventure partner I could have hoped for. I am looking forward to our next adventure!

I am profoundly grateful to my family: my parents, Gabriela and Nicolae, my grandmother, Irina (to whom this thesis is dedicated), my aunt, Mihaela, and my cousin, Silvana. Thank your for your constant, loving support, not only during the PhD, but throughout my entire education. You have always seen in me the best version of myself and given me courage to believe in that version too.

My work has been supported in part by the European Research Council (ERC) Grant 339539 (AOC).

Igor Zablotchi *Lausanne, 2020*

Preface

The research presented in this dissertation was conducted in the Distributed Computing Laboratory at EPFL, under the supervision of Professor Rachid Guerraoui, between 2015 and 2020. The main results of this dissertation appear originally in the following publications (author names are in alphabetical order):

1. Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, Athanasios Xygkis, Igor Zablotchi. Microsecond Consensus for Microsecond Applications. *Under submission.*

2. Rachid Guerraoui, Alex Kogan, Virendra Marathe, Igor Zablotchi. Efficient Multi-word Compare and Swap. In *International Symposium on Distributed Computing (DISC)*, 2020.

3. Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, Igor Zablotchi. The Impact of RDMA on Agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

4. Nachshon Cohen, Rachid Guerraoui, Igor Zablotchi. The Inherent Cost of Remembering Consistently. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

5. Oana Balmau, Rachid Guerraoui, Maurice Herlihy, Igor Zablotchi. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

Besides the above-mentioned publications, I was also involved in other research projects that resulted in the following publications:

1. Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, Igor Zablotchi. Log-Free Concurrent Data Structures. In *USENIX Annual Technical Conference (ATC)*, 2018.

2. Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, Igor Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *European Conference on Computer Systems (EuroSys)*, 2017

3. Peva Blanchard, Rachid Guerraoui, Julien Stainer, Igor Zablotchi. The Disclosure Power of Shared Objects. In *International Conference on Networked Systems (NETYS)*, 2017

Abstract

In this thesis, we revisit classic problems in shared-memory distributed computing through the lenses of (1) emerging hardware technologies and (2) changing requirements. Our contributions consist, on the one hand, in providing a better understanding of the fundamental benefits and limitations of new technologies, and on the other hand, in introducing novel, efficient tools and systems to ease the task of leveraging new technologies or meeting new requirements.

First, we look at Remote Direct Memory Access (RDMA), a networking hardware feature which enables a computer to access the memory of a remote computer without involving the remote CPU. In recent years, the distributed computing community has taken an interest in RDMA due to its ultra-low latency and high throughput and has designed systems that take advantage of these characteristics. However, we argue that the potential of RDMA for distributed computing remains largely untapped. We show that RDMA's unique semantics enable agreement algorithms which improve on fundamental trade-offs in distributed computing between performance and failure-tolerance. Furthermore, we show the practical applicability of our theoretical results through *Mu*, a state machine replication system which can replicate requests in under 2 microseconds, and can fail-over in under 1 millisecond when failures occur. Mu's replication and fail-over latencies are at least 61% and 90% lower, respectively, than those of prior work.

Second, we focus on persistent memory, a novel class of memory technologies which is only now starting to become available. Persistent memory provides byte-addressable persistent storage with access times comparable to traditional DRAM. Recent work has focused on designing tools for working with persistent memory, but little is known about the fundamental cost of providing consistency in persistent memory. Furthermore, important shared-memory primitives do not yet have efficient persistent implementations. We provide an answer to the former question through a tight bound on the number of persistent fences required to implement a lock-free persistent object. We address the latter problem by presenting a novel efficient multi-word compare-and-swap algorithm for persistent memory.

Third and finally, we consider the current exponential increase in the amount of data worldwide. Memory capacity has been on the rise for decades, but remains scarce when compared to the rate of data growth. Given this scarcity and the prevalence of concurrent in-memory processing, the classic problem of concurrent memory reclamation remains highly relevant

to this day. Previous work in this area has produced solutions which are either (a) fast but easily disrupted by process delays, or (b) slow but robust to process delays. We combine the best of both worlds in *QSense*, a memory reclamation algorithm which is fast in the common case when there are no process delays and falls back to a robust reclamation algorithm when process delays prevent the fast path from making progress.

Keywords: distributed algorithms, consensus, crash-fault tolerance, Byzantine-fault tolerance, state machine replication, microsecond computing, Remote Direct Memory Access (RDMA), persistent memory, multi-word compare-and-swap, memory reclamation

Résumé

Dans cette thèse, nous revisitons les problèmes classiques du calcul distribué à mémoire partagée à travers le prisme (1) des technologies émergentes et (2) de l'évolution des exigences. Nos contributions consistent, d'une part, à mieux comprendre les avantages et les limites fondamentales des nouvelles technologies et, d'autre part, à introduire des outils nouveaux et efficaces pour faciliter l'exploitation des nouvelles technologies ou la satisfaction de nouvelles exigences.

Nous examinons tout d'abord l'accès direct à la mémoire à distance (RDMA), une fonctionnalité du matériel de réseau qui permet à un ordinateur d'accéder à la mémoire d'un ordinateur distant sans impliquer le CPU distant. Ces dernières années, la communauté du calcul distribuée s'est intéressée à la RDMA en raison de sa latence faible et de son débit élevé, et a conçu des systèmes qui tirent parti de ces caractéristiques. Cependant, nous soutenons que le potentiel de la RDMA reste largement inexploité. Nous montrons que la sémantique unique de la RDMA permet des algorithmes de consensus qui améliorent les compromis fondamentaux dans le calcul distribué entre la performance et la tolérance aux pannes. En outre, nous montrons l'applicabilité pratique de nos résultats théoriques grâce à Mu, un système de réplication de machines à états qui répond aux requêtes en moins de 2 microsecondes, et bascule en moins d'une milliseconde en cas de défaillance.

Deuxièmement, nous nous concentrons sur la mémoire persistante, une nouvelle classe de technologies de mémoire qui offrent un stockage persistant adressable par octet avec des temps d'accès comparables à ceux de la mémoire volatile. Les travaux récents se sont concentrés sur la conception d'outils pour travailler avec la mémoire persistante, mais on sait encore peu de choses sur le coût de la cohérence en mémoire persistante. En outre, d'importantes primitives de mémoire partagée ne disposent pas encore de mises en œuvre efficaces pour la mémoire persistante. Nous apportons une réponse à la première question par le biais d'une limite stricte sur le nombre de barrières persistantes requises pour mettre en œuvre un objet persistant. Nous abordons le second problème en présentant un nouvel algorithme efficace pour le problème de la comparaison et remplacement de plusieurs mots pour la mémoire persistante.

Enfin, nous examinons l'augmentation exponentielle actuelle de la quantité de données dans le monde. La capacité de la mémoire augmente depuis des décennies, mais elle reste limitée par rapport au taux de croissance des données. Compte tenu de cette disparité et de la

prévalence du traitement concurrent des données en mémoire, le problème classique de la récupération concurrente de la mémoire reste très pertinent à ce jour. Les travaux antérieurs dans ce domaine ont produit des solutions qui sont soit (a) rapides mais facilement perturbées par les retards des processus, soit (b) lentes mais résistantes aux retards. Nous combinons le meilleur des deux mondes dans QSense, un algorithme de récupération de la mémoire qui est rapide dans le cas commun où il n'y a pas de retard de processus, et qui bascule à un algorithme de récupération robuste lorsque les retards de processus empêchent la voie rapide de progresser.

Mots clés : consensus, tolérance aux pannes, tolérance aux pannes byzantines, réplication de machines à états, accès direct à la mémoire à distance (RDMA), mémoire persistante, comparaison et échange de plusieurs mots, récupération de mémoire

Contents

Acknowledgements **i**

Preface **iii**

Abstract (English/Français) **v**

List of Figures **xv**

List of Tables **xvii**

List of Algorithms and Listings **xix**

1 Introduction **1**
- 1.1 Contributions 2
  - 1.1.1 Shared remote memory 2
  - 1.1.2 Shared persistent memory 3
  - 1.1.3 Memory reclamation 4
- 1.2 Thesis Roadmap 4

I Sharing Remote Memory **7**

2 The Impact of RDMA on Agreement **9**
- 2.1 Introduction 9
- 2.2 Related Work 12
- 2.3 Model and Preliminaries 14
- 2.4 Byzantine Failures 17
  - 2.4.1 The Robust Backup Sub-Algorithm 18
  - 2.4.2 The Cheap Quorum Sub-Algorithm 23
  - 2.4.3 Putting it Together: the Fast & Robust Algorithm 25
- 2.5 Crash Failures 27
- 2.6 Dynamic Permissions are Necessary for Efficient Consensus 29
- 2.7 RDMA in Practice 30

3 Microsecond Consensus for Microsecond Applications **33**
- 3.1 Introduction 33

3.2 Background 36
3.2.1 Microsecond Applications and Computing 36
3.2.2 State Machine Replication 37
3.2.3 RDMA 37
3.3 Overview of Mu 38
3.3.1 Architecture 38
3.3.2 RDMA Communication 40
3.4 Replication Plane 40
3.4.1 Basic Algorithm 41
3.4.2 Extensions 43
3.5 Background Plane 44
3.5.1 Leader Election 44
3.5.2 Permission Management 45
3.5.3 Log Recycling 46
3.5.4 Adding and Removing Replicas 46
3.6 Implementation 47
3.7 Evaluation 47
3.7.1 Common-Case Latency 48
3.7.2 Fail-Over Time 51
3.7.3 Throughput 52
3.8 Related Work 54
3.9 Conclusion 56

II Sharing Persistent Memory **57**

4 The Inherent Cost of Remembering Consistently **59**
4.1 Introduction 59
4.2 Background 62
4.2.1 Persistent Memory 62
4.2.2 Processes and Operations 63
4.3 ONLL: a Primer 63
4.3.1 Rationale 64
4.3.2 ONLL Design 64
4.3.3 Illustration: Shared Counter 65
4.4 ONLL: a Universal Construction 67
4.4.1 Data Structures 67
4.4.2 ONLL Algorithm 69
4.5 ONLL: Correctness 71
4.5.1 Lock-freedom 71
4.5.2 Durable linearizability 72
4.6 Lower Bound 77
4.7 Related Work 79

4.8 Concluding Remarks . . . 81

5 Efficient Multi-Word Compare-and-Swap 83
5.1 Introduction . . . 83
5.2 System Model . . . 86
5.2.1 Volatile Memory . . . 86
5.2.2 Persistent Memory . . . 87
5.3 Impossibility . . . 88
5.4 Volatile MCAS with $k+1$ CAS . . . 91
5.4.1 High-level Description . . . 91
5.4.2 Technical Details . . . 91
5.4.3 Correctness . . . 94
5.5 Persistent MCAS with $k+1$ CAS and 2 Persistent Fences . . . 98
5.6 Memory Management . . . 100
5.6.1 Managing Persistent Memory . . . 101
5.6.2 Efficient Reads . . . 102
5.7 Evaluation . . . 102
5.7.1 Experimental Setup . . . 102
5.7.2 Array Benchmark . . . 103
5.7.3 Doubly-linked List Benchmark . . . 106
5.7.4 B+-tree Benchmark . . . 106
5.8 Related Work . . . 107
5.9 Conclusion . . . 110

III Memory Reclamation 111

6 Fast and Robust Memory Reclamation for Concurrent Data Structures 113
6.1 Introduction . . . 113
6.1.1 The Problem . . . 113
6.1.2 The Trade-off . . . 114
6.1.3 The Contributions . . . 115
6.2 Model and Problem Definition . . . 117
6.2.1 Node States . . . 117
6.2.2 The Memory Reclamation Problem . . . 117
6.2.3 Terminology . . . 118
6.3 Background . . . 118
6.3.1 Quiescent State Based Reclamation . . . 118
6.3.2 Hazard Pointers (HP) . . . 119
6.4 An Overview of QSense . . . 121
6.4.1 Rationale . . . 121
6.4.2 QSense in a Nutshell . . . 122
6.5 Cadence . . . 123

6.5.1 The Fallback Path . . . 124
6.5.2 Merging the Fast and Fallback Paths . . . 127
6.6 Correctness & Complexity . . . 129
6.6.1 Cadence . . . 130
6.6.2 QSense . . . 130
6.7 Experimental Evaluation . . . 132
6.7.1 Experimental Setting . . . 132
6.7.2 Methodology . . . 132
6.7.3 Results . . . 133
6.8 Related Work . . . 134
6.9 Conclusion . . . 136

IV Concluding Remarks 137

7 Conclusions and Future Work 139
7.1 Summary and Implications . . . 139
7.2 Future Directions . . . 140

V Appendices 143

A The Impact of RDMA on Agreement 145
A.1 Correctness of Reliable Broadcast . . . 145
A.2 Correctness of Cheap Quorum . . . 147
A.3 Correctness of the Fast & Robust . . . 149
A.4 Correctness of Protected Memory Paxos . . . 151

B Microsecond Consensus for Microsecond Applications 155
B.1 Pseudocode of the Basic Version . . . 155
B.2 Definitions . . . 156
B.3 Invariants . . . 156
B.3.1 Validity . . . 156
B.3.2 Agreement . . . 157
B.3.3 Termination . . . 158
B.4 Optimizations & Additions . . . 159
B.4.1 New Leader Catch-Up . . . 159
B.4.2 Update Followers . . . 160
B.4.3 Followers Update Their Own FUO . . . 161
B.4.4 Grow Confirmed Followers . . . 162
B.4.5 Omit Prepare Phase . . . 163

C The Inherent Cost of Remembering Consistently 165

D Efficient Multi-Word Compare-and-Swap **167**
D.1 Replacing RDCSS with CAS in Harris et al. algorithm leads to ABA 167
D.2 Performance in read-only and update-heavy workloads 167

E Fast and Robust Memory Reclamation for Concurrent Data Structures **171**
E.1 QSBR Correctness 171
E.2 QSense on a Linked List 172

Bibliography **177**

Curriculum Vitae **195**

List of Figures

2.1 Our message-and-memory model with permissions. 14
2.2 Interactions of the components of the Fast & Robust Algorithm. 26

3.1 Architecure of Mu. 38
3.2 Performance comparison of different RDMA permission switching mechanisms. 46
3.3 Replication latency of Mu integrated into different applications, with varying payload sizes. 49
3.4 Replication latency of Mu compared with other replication solutions. 50
3.5 End-to-end latencies of applications, with and without replication. 51
3.6 Fail-over time distribution in Mu. 52
3.7 Latency vs throughput in Mu. 53

4.1 Executions of a counter implemented using ONLL. 66
4.2 The execution trace and the fuzzy window in ONLL. 68

5.1 Array benchmark for MCAS. 104
5.2 Doubly-linked list benchmark for MCAS (80% and 98% reads). 105
5.3 B+-tree benchmark for MCAS (80% and 98% reads). 105

6.1 The concurrent memory reclamation problem. 114
6.2 A high-level view of QSense. 116
6.3 QSense, HP and no reclamation on a linked list of 2000 elements, with a 10% updates workload. 117
6.4 Rooster processes and deferred reclamation. 126
6.5 Time diagram for the proof of QSense liveness. 131
6.6 Scalability of memory reclamation on a linked list (2000 elements), a skip list (20000 elements) and a BST (2000000 elements) with 50% updates. 133
6.7 Path switching with process delays (8 processes, 50% updates). 134

C.1 Illustrating a read from a node that is never the latest in the non-fuzzy part. . . 166

D.1 Doubly-linked list benchmark for MCAS (50% and 100% reads). 168
D.2 B+-tree benchmark for MCAS (50% and 100% reads). 169

List of Tables

2.1 Known fault tolerance results for Byzantine agreement. 13

5.1 Comparison of non-blocking MCAS implementations. 108

List of Algorithms and Listings

2.1 Reliable Broadcast. 19
2.2 Validate Operation for Reliable Broadcast. 22
2.3 Cheap Quorum normal operation—code for process p. 24
2.4 Cheap Quorum panic mode—code for process p. 24
2.5 Preferential Paxos—code for process p. 25
2.6 Protected Memory Paxos—code for process p. 28

3.1 Log Structure. 41
3.2 Basic Replication Algorithm of Mu. 42

4.1 The recordEntry structure used by ONLL. 68
4.2 The execution trace used by ONLL. 70
4.3 The update operation in ONLL. 70
4.4 The read operation in ONLL. 71
4.5 Recovery in ONLL. 71

5.1 Data structures used by our MCAS algorithm for volatile memory. 92
5.2 The `readInternal` auxiliary function, used by our MCAS algorithm for volatile memory. 92
5.3 Our MCAS algorithm for volatile memory. 93
5.4 The `readInternal` auxiliary function, used by our MCAS algorithm for persistent memory. 98
5.5 Our MCAS algorithm for persistent memory. 99

6.1 High-level steps taken when accessing a node in a data structure using *hazard pointers*. 120
6.2 Example of illegal operation interleaving. 122
6.3 Main functions of Cadence. 125
6.4 High-level steps taken when accessing a node in a data structure using *Cadence*. 126
6.5 Main QSense functions (I). 128
6.6 Main QSense functions (II). 129

B.1 Optimization: Leader Catch Up. 160
B.2 Optimization: Update Followers. 160

B.3 Optimization: Followers Update Their Own FUO. 161

E.1 QSense on a concurrent linked-list (I). 173
E.2 QSense on a concurrent linked-list (II). 174
E.3 QSense on a concurrent linked-list (III). 175

1 Introduction

Distributed computing is concerned with devising means for multiple processes to collaborate on some common task, typically with the aim of improving performance or failure-tolerance, when compared to solutions which only employ a single process. Traditionally, in distributed computing, a system is modeled as either message-passing (if processes are distinct computers which communicate by exchanging messages over the network) or shared-memory (if processes are distinct threads of execution inside a computer, which communicate by performing operations on a common memory space).

Historically, distributed computing has been an extensively studied area, with research going back at least five decades [79]. Despite this wealth of research (or perhaps because of it), distributed computing remains a continually evolving field, both due to changing requirements and due to technological advancements which are changing the fundamental tools we have at our disposal in order to achieve those requirements. Here we identify three key areas—all revolving around the theme of modern shared memory—which have either not been well studied so far, or for which existing solutions are inefficient.

Remote Direct Memory Access (RDMA). RDMA allows a computer to access the memory of a remote computer, without CPU involvement at the remote side. Furthermore, RDMA provides low-latency communication by bypassing the OS kernel and by implementing several layers of the network stack in hardware. RDMA enables us to think of physically distinct machines as simultaneously passing messages and sharing memory—a mix of the message passing and shared memory models. In recent years, research has focused on exploiting RDMA's better raw performance (low latency, high throughput) to build faster distributed systems: key-value stores [84, 132], remote procedure call frameworks [131], and state machine replication (SMR) systems [199, 225], among others. Despite this excellent previous work, we argue that a direction which has not been well studied so far is the potential for RDMA's unique semantics to unlock fundamentally lower-complexity (and thus faster) distributed algorithms.

Persistent memory. Persistent memory (also called non-volatile RAM, NVRAM or NVM) is fast, byte-addressable memory that preserves its contents even in the absence of power. Its access times are comparable to those of traditional (volatile) DRAM, thus making it a suitable candidate for main memory. This raises the prospect of a new class of objects: in-memory persistent objects. Of particular interest are concurrent implementations of such objects, in which several processes concurrently perform operations in order to increase performance. Such implementations are desirable because they enable much faster operation and recovery, when persistence is required, as compared to implementations that make use of traditional stable storage. However, concurrent implementations of persistent objects face the challenge of correct recovery after a crash, without negating the performance benefits on the common path. Recent work has focused on defining suitable safety criteria [29, 100, 127], developing fast and scalable data structures [57, 91, 153, 189, 229], and building transactional frameworks that provide generic all-or-nothing semantics for interacting with persistent memory [33, 55, 63, 75, 96, 126, 138, 141, 163, 175, 224]. Yet, the fundamental (lower-bound) cost of implementing concurrent persistent objects has not been well studied to date; moreover, many foundational shared-memory primitives do not yet have efficient implementations for persistent memory.

Memory reclamation. The global amount of digital data created, captured or replicated is growing exponentially: it was estimated at 33 Zettabytes in 2018 and it is expected to grow to 175 Zettabytes by 2025 [207]. The global amount of main memory remains comparatively scarce. Given this scarcity and the increasing trend towards in-memory processing on multicore machines, memory reclamation remains an important problem in modern shared-memory distributed computing. Despite decades of research, existing memory reclamation schemes are either robust to process delays but slow (e.g., hazard pointers), fast but not robust (e.g., epoch-based reclamation) or both fast and robust, but only narrowly applicable.

1.1 Contributions

In this thesis, we explore the opportunities and challenges of distributed computing with modern shared memory, as raised by the three aforementioned areas. We describe our contributions in more detail below (contributions are highlighted using $\star$).

1.1.1 Shared remote memory

Here we focus on solving the problem of consensus, in which processes need to agree on a common value, in an RDMA-enabled system. Consensus is of central importance in distributed computing, because it enables applications to be replicated across multiple hosts (called *replicas*) and thus provide high availability, i.e., continue to be responsive even if a subset of the replicas fail. We consider two types of failures: (1) crash failures (in which faulty replicas permanently cease to take steps) and (2) Byzantine failures (in which faulty replicas may deviate arbitrarily—even maliciously—from the protocol). We measure failure-tolerance

by the lowest total number of replicas required to ensure correctness if up to f replicas can be faulty. We measure performance as the number of network delays required to solve consensus in the best case; a message is counted as one network delay, and a one-sided RDMA operation (i.e., read or write) is counted as two network delays (one for the invocation and one for the response).

- ⋆ We show that RDMA improves on the fundamental trade-off between failure-tolerance and performance in consensus algorithms. Specifically, we introduce a theoretical model that captures the key properties of RDMA and show that in our model:
  - (a) Byzantine agreement can be solved with $2f+1$ replicas in 2 network delays under standard eventual synchrony assumptions; it is known from previous work that this is impossible in a purely message-passing model.
  - (b) Crash-fault tolerant consensus can be solved with $f+1$ replicas in 2 network delays under standard eventual synchrony assumptions; previous results in the message-passing and shared-memory models require either $2f+1$ replicas or 4 delays, respectively.
- ⋆ We introduce *Mu*, the first state machine replication system that requires a single one-sided RDMA operation (a remote write) to replicate a request in the common case. Mu takes less than 1.3 microseconds to replicate a (small) request, and less than a millisecond to fail-over in case of failure—cutting replication and failure recovery latencies of prior systems by at least 61% and 90%, respectively.

1.1.2 Shared persistent memory

Here we focus on *lock-free* implementations of persistent objects. Lock-free implementations are desirable because they guarantee that the system overall makes progress despite arbitrary process delays (as long as some process takes steps). An important metric when considering concurrent implementations of persistent objects is the number of persistence fences used. These are expensive primitives which allow programmers to control the order in which read and write operations reach persistent memory.

- ⋆ We provide a tight bound on the number of persistence fences required for any lock-free deterministic implementation of a persistent object.

We also consider the problem of efficiently implementing atomic primitives for persistent memory. Compare-and-swap (CAS) is perhaps the best known such primitive; it is used pervasively in lock-free algorithms. CAS conditionally updates a memory location, provided that its contents match some expected value. We focus here on the multi-word compare-and-swap (MCAS) primitive, a natural generalization of CAS that acts on multiple locations atomically. MCAS can significantly simplify algorithms which need to operate atomically

on multiple variables at the same time (e.g., doubly-linked lists [198], B+-trees [17, 46]). As before, we focus on lock-free implementations and we additionally consider the desirable disjoint-access-parallelism property, which ensures that MCAS calls that operate on disjoint sets of memory locations do not conflict with each other.

- ⋆ We introduce a novel multi-word compare-and-swap (MCAS) algorithm for persistent and volatile memory. Our algorithm is more efficient than state-of-the-art MCAS algorithms: it is the first to use only $k+1$ CASes for a k-word MCAS; existing solutions require at least $2k+1$ CASes. The persistent version of our algorithm requires only 2 persistence fences per call to MCAS, as opposed to $2k+1$ for the best previous result. We additionally prove that both versions are almost optimal in terms of the number of CASes used, by providing a lower bound on the number of CAS instructions required for any lock-free disjoint-access-parallel implementation of MCAS.

1.1.3 Memory reclamation

We focus on memory reclamation for lock-free data structures, in programming languages without automatic garbage collection (such as C or C++). In this context, the problem of memory reclamation can be stated as follows: given one or more data structure nodes that have been removed from the data structure, determine when it is safe to free or reuse those nodes' memory.

- ⋆ We introduce *QSense*, a novel memory reclamation scheme with a hybrid design, consisting of a fast path and a slow path. In the common case when there are no process delays, QSense uses a fast, blocking, epoch-based reclamation scheme. When process delays prevent the fast path from making progress, QSense switches to a robust fall-back path—a novel variant of hazard pointers which virtually eliminates the need for expensive memory fences.

1.2 Thesis Roadmap

The rest of the thesis is organized in five parts, consisting of seven chapters and five appendices, as follows.

Part I

- In Chapter 2 we show that for crash- and Byzantine-fault tolerant consensus, RDMA enables a better trade-off between performance and failure-tolerance than either the message-passing or the shared-memory models.
- Chapter 3 introduces Mu, the first RDMA-based state machine replication system suitable for applications with microsecond latencies.

Part II	• Chapter 4 gives the first tight bound on the number of persistence fences required to implement a shared persistent object in the lock-free, deterministic case. • Chapter 5 introduces a novel and efficient multi-word compare-and-swap algorithm for volatile and persistent memory.
Part III	• Chapter 6 introduces QSense, a fast and robust concurrent memory reclamation scheme.
Part IV	• Chapter 7 concludes the thesis with a summary of contributions and their implications, as well as an outline of future research directions.
Part V	• Appendices A through E contain supplementary material (proofs, extensions, examples, graphs etc.) related to Chapters 2–6, respectively.

Sharing Remote Memory Part I

2 The Impact of RDMA on Agreement

Remote Direct Memory Access (RDMA) is becoming widely available in data centers. This technology allows a process to directly read and write the memory of a remote host, with a mechanism to control access permissions. In this chapter, we study the fundamental power of these capabilities. We consider the well-known problem of achieving consensus despite failures, and find that RDMA can improve the inherent trade-off in distributed computing between failure resilience and performance. Specifically, we show that RDMA allows algorithms that simultaneously achieve high resilience and high performance, while traditional algorithms had to choose one or another. With Byzantine failures, we give an algorithm that only requires $n \geq 2f_P + 1$ processes (where f_P is the maximum number of faulty processes) and decides in two (network) delays in common executions. With crash failures, we give an algorithm that only requires $n \geq f_P + 1$ processes and also decides in two delays. Both algorithms tolerate a minority of memory failures inherent to RDMA, and they provide safety in asynchronous systems and liveness with standard additional assumptions.

2.1 Introduction

In recent years, a technology known as Remote Direct Memory Access (RDMA) has made its way into data centers, earning a spotlight in distributed systems research. RDMA provides the traditional send/receive communication primitives, but also allows a process to directly read/write remote memory. Research work shows that RDMA leads to some new and exciting distributed algorithms [5, 84, 129, 132, 199, 225].

RDMA provides a different interface from previous communication mechanisms, as it combines message-passing with shared-memory [5]. Furthermore, to safeguard the remote memory, RDMA provides *protection* mechanisms to grant and revoke access for reading and writing data. This mechanism is fine grained: an application can choose subsets of remote memory called *regions* to protect; it can choose whether a region can be read, written, or both; and it can choose individual processes to be given access, where different processes can have different accesses. Furthermore, protections are *dynamic*: they can be changed by the application

over time. In this chapter, we lay the groundwork for a theoretical understanding of these RDMA capabilities, and we show that they lead to distributed algorithms that are inherently more powerful than before.

While RDMA brings additional power, it also introduces some challenges. With RDMA, the remote memories are subject to failures that cause them to become unresponsive. This behavior differs from traditional shared memory, which is often assumed to be reliable[1]. In this chapter, we show that the additional power of RDMA more than compensates for these challenges.

Our main contribution is to show that RDMA improves on the fundamental trade-off in distributed systems between failure resilience and performance—specifically, we show how a consensus protocol can use RDMA to achieve *both* high resilience and high performance, while traditional algorithms had to choose one or another. We illustrate this on the fundamental problem of achieving consensus and capture the above RDMA capabilities as an M&M model [5], in which processes can use both message-passing and shared-memory. We consider asynchronous systems and require safety in all executions and liveness under standard additional assumptions (e.g., partial synchrony). We measure resiliency by the number of failures an algorithm tolerates, and performance by the number of (network) delays in common-case executions. Failure resilience and performance depend on whether processes fail by crashing or by being Byzantine, so we consider both.

With Byzantine failures, we consider the consensus problem called weak Byzantine agreement, defined by Lamport [145]. We give an algorithm that (a) requires only $n \geq 2f_P + 1$ processes (where f_P is the maximum number of faulty processes) and (b) decides in two delays in the common case. With crash failures, we give the first algorithm for consensus that requires only $n \geq f_P + 1$ processes and decides in two delays in the common case. With both Byzantine or crash failures, our algorithms can also tolerate crashes of memory—only $m \geq 2f_M + 1$ memories are required, where f_M is the maximum number of faulty memories.

Our algorithms appear to violate known impossibility results: it is known that with message-passing, Byzantine agreement requires $n \geq 3f_P + 1$ even if the system is synchronous [194], while consensus with crash failures require $n \geq 2f_P + 1$ if the system is partially synchronous [87]. There is no contradiction: our algorithms rely on the power of RDMA, not available in other systems.

RDMA's power comes from two features: (1) simultaneous access to message-passing and shared-memory, and (2) dynamic permissions. Intuitively, shared-memory helps resilience, message-passing helps performance, and dynamic permissions help both.

To see how shared-memory helps resilience, consider the Disk Paxos algorithm [92], which uses shared-memory (disks) but no messages. Disk Paxos requires only $n \geq f_P + 1$ processes, matching the resilience of our algorithm. However, Disk Paxos is not as fast: it takes at least

[1] There are a few studies of failure-prone memory, as we discuss in related work.

four delays. In fact, we show that no shared-memory consensus algorithm can decide in two delays (Section 2.6).

To see how message-passing helps performance, consider the Fast Paxos algorithm [149], which uses message-passing and no shared-memory. Fast Paxos decides in only two delays in common executions, but it requires $n \geq 2f_P + 1$ processes.

Of course, the challenge is achieving both high resilience and good performance in a single algorithm. This is where RDMA's dynamic permissions shine. Clearly, dynamic permissions improve resilience against Byzantine failures, by preventing a Byzantine process from overwriting memory and making it useless. More surprising, perhaps, is that dynamic permissions help performance, by providing an uncontended instantaneous guarantee: if each process revokes the write permission of other processes before writing to a register, then a process that writes successfully knows that it executed uncontended, without having to take additional steps (e.g., to read the register). We use this technique in our algorithms for both Byzantine and crash failures.

In summary, our contributions are as follows:

- We consider distributed systems with RDMA, and we propose a model that captures some of its key properties while accounting for failures of processes and memories, with support of dynamic permissions.
- We show that the shared-memory part of our RDMA improves resilience: our Byzantine agreement algorithm requires only $n \geq 2f_P + 1$ processes.
- We show that the shared-memory by itself does not permit consensus algorithms that decide in two steps in common executions.
- We show that with dynamic permissions, we can improve the performance of our Byzantine Agreement algorithm, to decide in two steps in common executions.
- We give similar results for the case of crash failures: decision in two steps while requiring only $n \geq f_P + 1$ processes.
- Our algorithms can tolerate the failure of memories, up to a minority of them.

The rest of the chapter is organized as follows. Section 2.2 gives an overview of related work. In Section 2.3 we formally define the RDMA-compliant M&M model that we use in the rest of the chapter, and specify the agreement problems that we solve. Section 2.4 presents our fast and resilient Byzantine agreement algorithm. In Section 2.5 we consider the special case of crash-only failures, and show an improvement of the algorithm and tolerance bounds for this setting. In Section 2.6 we briefly outline a lower bound that shows that the dynamic permissions of RDMA are necessary for achieving our results. Finally, in Section 2.7 we discuss the semantics of RDMA in practice, and how our model reflects these features. To ease readability, most proofs have been deferred to Appendix A.

2.2 Related Work

RDMA. Many high-performance systems were recently proposed using RDMA, such as distributed key-value stores [84, 132], communication primitives [84, 134], and shared address spaces across clusters [84]. Kalia et al. [133] provide guidelines for designing systems using RDMA. RDMA has also been applied to solve consensus [129, 199, 225]. Our model shares similarities with DARE [199] and APUS [225], which modify queue-pair state at run time to prevent or allow access to memory regions, similar to our dynamic permissions. These systems perform better than TCP/IP-based solutions, by exploiting better raw performance of RDMA, without changing the fundamental communication complexity or failure-resilience of the consensus protocol. Similarly, Rüsch et al. [209] use RDMA as a replacement for TCP/IP in existing BFT protocols.

M&M. Message-and-memory (M&M) refers to a broad class of models that combine message-passing with shared-memory, introduced by Aguilera et al. in [5]. In that work, Aguilera et al. consider M&M models without memory permissions and failures, and show that such models lead to algorithms that are more robust to failures and asynchrony. In particular, they give a consensus algorithm that tolerates more crash failures than message-passing systems, but is more scalable than shared-memory systems, as well as a leader election algorithm that reduces the synchrony requirements. In this chapter, our goal is to understand how memory permissions and failures in RDMA impact agreement.

Byzantine Fault Tolerance. Lamport, Shostak and Pease [150, 194] show that Byzantine agreement can be solved in synchronous systems iff $n \geq 3f_P + 1$. With unforgeable signatures, Byzantine agreement can be solved iff $n \geq 2f_P + 1$. In asynchronous systems subject to failures, consensus cannot be solved [89]. However, this result is circumvented by making additional assumptions for liveness, such as randomization [28] or partial synchrony [54, 87]. Many Byzantine agreement algorithms focus on safety and implicitly use the additional assumptions for liveness. Even with signatures, asynchronous Byzantine agreement can be solved only if $n \geq 3f_P + 1$ [39].

It is well known that the resilience of Byzantine agreement varies depending on various model assumptions like synchrony, signatures, equivocation, and the exact variant of the problem to be solved. A system that has non-equivocation is one that can prevent a Byzantine process from sending different values to different processes. Table 2.1 summarizes some known results that are relevant to this chapter.

Our Byzantine agreement results share similarities with results for shared memory. Malkhi et al. [167] and Alon et al. [10] show bounds on the resilience of strong and weak consensus in a model with reliable memory but Byzantine processes. They also provide consensus protocols, using read-write registers enhanced with sticky bits (write-once memory) and access control lists not unlike our permissions. Bessani et al. [31] propose an alternative to sticky bits and

Work	Synchrony	Signatures	Non-Equiv	Strong Validity	Resiliency
[150]	✓	✓	✗	✓	$2f+1$
[150]	✓	✗	✗	✓	$3f+1$
[10, 167]	✗	✓	✓	✓	$3f+1$
[60]	✗	✓	✗	✗	$3f+1$
[60]	✗	✗	✓	✗	$3f+1$
[60]	✗	✓	✓	✗	$2f+1$
This chapter	✗	✓	✗ (RDMA)	✗	$2f+1$

Table 2.1 – Known fault tolerance results for Byzantine agreement.

access control lists through Policy-Enforced Augmented Tuple Spaces. All these works handle Byzantine failures with powerful objects rather than registers. Bouzid et al. [37] show that $3f_P+1$ processes are necessary for strong Byzantine agreement with read-write registers.

Some prior work solves Byzantine agreement with $2f_P+1$ processes using specialized trusted components that Byzantine processes cannot control [58, 59, 68, 69, 135, 223]. Some schemes decide in two delays but require a large trusted component: a coordinator [58], reliable broadcast [69], or message ordering [135]. For us, permission checking in RDMA is a trusted component of sorts, but it is small and readily available.

At a high-level, our improved Byzantine fault tolerance is achieved by preventing equivocation by Byzantine processes, thereby effectively translating each Byzantine failure into a crash failure. Such translations from one type of failure into a less serious one have appeared extensively in the literature [27, 39, 60, 185]. Early work [27, 185] shows how to translate a crash tolerant algorithm into a Byzantine tolerant algorithm in the synchronous setting. Bracha [38] presents a similar translation for the asynchronous setting, in which $n \geq 3f_P+1$ processes are required to tolerate f_P Byzantine failures. Bracha's translation relies on the definition and implementation of a reliable broadcast primitive, very similar to the one in this chapter. However, we show that using the capabilities of RDMA, we can implement it with higher fault tolerance.

Faulty memory. Afek et al. [3] and Jayanti et al. [128] study the problem of masking the benign failures of shared memory or objects. We use their ideas of replicating data across memories. Abraham et al. [1] considers honest processes but malicious memory.

Common-case executions. Many systems and algorithms tolerate adversarial scheduling but optimize for common-case executions without failures, asynchrony, contention, etc (e.g., [35, 80, 86, 137, 143, 149, 168]). None of these match both the resilience and performance of

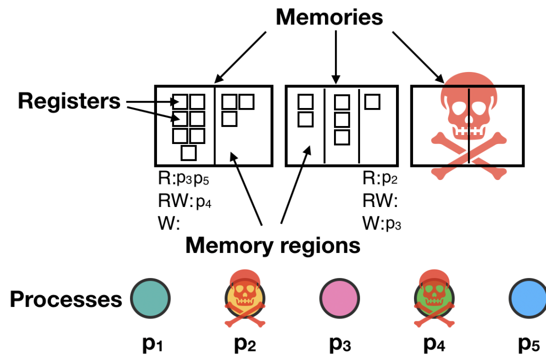


Figure 2.1 – Our model with processes and memories, which may both fail. Processes can send messages to each other or access registers in the memories. Registers in a memory are grouped into memory regions that may overlap. Each region has a permission indicating what processes can read, write, and read-write the registers in the region (shown for two regions).

our algorithms. Some algorithms decide in one delay but require $n \geq 5f_P + 1$ for Byzantine failures [217] or $n \geq 3f_P + 1$ for crash failures [42, 80].

2.3 Model and Preliminaries

We consider a message-and-memory (M&M) model (Figure 2.1), which allows processes to use both message-passing and shared-memory [5]. The system has n processes $P = \{p_1, \ldots, p_n\}$ and m (shared) *memories* $M = \{\mu_1, \ldots, \mu_m\}$. Processes communicate by accessing memories or sending messages. Throughout the chapter, memory refers to the shared memories, not the local state of processes.

The system is asynchronous in that it can experience arbitrary delays. We expect algorithms to satisfy the safety properties of the problems we consider, under this asynchronous system. For liveness, we require additional standard assumptions, such as partial synchrony, randomization, or failure detection.

Memory permissions. Each memory consists of a set of *registers*. To control access, an algorithm groups those registers into a set of (possibly overlapping) *memory regions*, and then defines permissions for those memory regions. Formally, a memory region mr of a memory μ is a subset of the registers of μ. We often refer to mr without specifying the memory μ explicitly. Each memory region mr has a *permission*, which consists of three disjoint sets of processes R_{mr}, W_{mr}, RW_{mr} indicating whether each process can read, write, or read-write the registers in the region. We say that *p has read permission on mr* if $p \in R_{mr}$ or $p \in RW_{mr}$; we say that *p has write permission on mr* if $p \in W_{mr}$ or $p \in RW_{mr}$. In the special case when $R_{mr} = P \setminus \{p\}$, $W_{mr} = \emptyset$, $RW_{mr} = \{p\}$, we say that mr is a Single-Writer Multi-Reader (SWMR) region—registers in mr correspond to the traditional notion of SWMR registers. Note that a

register may belong to several regions, and a process may have access to the register on one region but not another—this models the existing RDMA behavior. Intuitively, when reading or writing data, a process specifies the region and the register, and the system uses the region to determine if access is allowed (we make this precise below).

Permission change. An algorithm indicates an initial permission for each memory region *mr*. Subsequently, the algorithm may wish to change the permission of *mr* during execution. For that, processes can invoke an operation $changePermission(mr, new_perm)$, where *new_perm* is a triple (R, W, RW). This operation returns no results and it is intended to modify R_{mr}, W_{mr}, RW_{mr} to R, W, RW. To tolerate Byzantine processes, an algorithm can restrict processes from changing permissions. For that, the algorithm specifies a function $legalChange(p, mr, old_perm, new_perm)$ which returns a boolean indicating whether process p can change the permission of *mr* to *new_perm* when the current permissions are *old_perm*. More precisely, when *changePermission* is invoked, the system evaluates *legalChange* to determine whether *changePermission* takes effect or becomes a no-op. When *legalChange* always returns false, we say that the *permissions are static*; otherwise, the *permissions are dynamic*.

Accessing memories. Processes access the memories via $write(mr, r, v)$ and $read(mr, r)$ operations, for memory region *mr*, register r, and value v. A $write(mr, r, v)$ by process p changes register r to v and returns *ack* if $r \in mr$ and p has write permission on *mr*; otherwise, the operation returns *nak*. A $read(mr, r)$ by process p returns the last value successfully written to r if $r \in mr$ and p has read permission on *mr*; otherwise, the operation returns *nak*. In our algorithms, a register belongs to exactly one region, so we omit the *mr* parameter from write and read operations.

Sending messages. Processes can also communicate by sending messages over a set of directed links. We assume messages are unique. If there is a link from process p to process q, then p can send messages to q. Links satisfy two properties: *integrity* and *no-loss*. Given two correct processes p and q, integrity requires that a message m be received by q from p at most once and only if m was previously sent by p to q. No-loss requires that a message m sent from p to q be eventually received by q. In our algorithms, we typically assume a fully connected network so that every pair of correct processes can communicate. We also consider the special case when there are no links (see below).

Executions and steps. An execution is as a sequence of process steps. In each step, a process does the following, according to its local state: (1) sends a message or invokes an operation on a memory (read, write, or changePermission), (2) tries to receive a message or a response from an outstanding operation, and (3) changes local state. We require a process to have at most one outstanding operation on each memory.

Failures. A memory m may fail by crashing, which causes subsequent operations on its registers to hang without returning a response. Because the system is asynchronous, a process cannot differentiate a crashed memory from a slow one. We assume there is an upper bound f_M on the maximum number of memories that may crash. Processes may fail by crashing or becoming Byzantine. If a process crashes, it stops taking steps forever. If a process becomes Byzantine, it can deviate arbitrarily from the algorithm. However, that process cannot operate on memories without the required permission. We assume there is an upper bound f_P on the maximum number of processes that may be faulty. Where the context is clear, we omit the P and M subscripts from the number of failures, f.

Signatures. Our algorithms assume unforgeable signatures: there are primitives $sign(v)$ and $sValid(p, v)$ which, respectively, signs a value v and determines if v is signed by process p.

Messages and disks. The model defined above includes two common models as special cases. In the *message-passing* model, there are no memories ($m = 0$), so processes can communicate only by sending messages. In the *disk* model [92], there are no links, so processes can communicate only via memories; moreover, each memory has a single region which always permits all processes to read and write all registers.

Consensus

In the consensus problem, processes propose an initial value and must make an irrevocable decision on a value. With crash failures, we require the following properties:

- **Uniform Agreement.** If processes p and q decide v_p and v_q, then $v_p = v_q$.
- **Validity.** If some process decides v, then v is the initial value proposed by some process.
- **Termination.** Eventually all correct processes decide.

We expect Agreement and Validity to hold in an asynchronous system, while Termination requires standard additional assumptions (partial synchrony, randomization, failure detection, etc). With Byzantine failures, we change these definitions so the problem can be solved. We consider weak Byzantine agreement [145], with the following properties:

- **Agreement.** If correct processes p and q decide v_p and v_q, then $v_p = v_q$.
- **Validity.** With no faulty processes, if some process decides v, then v is the input of some process.
- **Termination.** Eventually all correct processes decide.

Complexity of algorithms. We are interested in the performance of algorithms in *common-case executions*, when the system is synchronous and there are no failures. In those cases, we measure performance using the notion of *delays*, which extends message-delays to our model. Under this metric, computations are instantaneous, each message takes one delay, and each memory operation (*write*, *read*, and *changePermission*) takes two delays. Intuitively, a delay represents the time incurred by the network to transmit a message; a memory operation takes two delays because its hardware implementation requires a round trip. We say that a consensus protocol is k-*deciding* if, in common-case executions, some process decides in k delays.

2.4 Byzantine Failures

We now consider Byzantine failures and give a 2-deciding algorithm for weak Byzantine agreement with $n \geq 2f_P + 1$ processes and $m \geq 2f_M + 1$ memories. The algorithm consists of the composition of two sub-algorithms: a slow one that always works, and a fast one that gives up under hard conditions.

The first sub-algorithm, called *Robust Backup*, is developed in two steps. We first implement a *reliable broadcast* primitive, which prevents Byzantine processes from sending different values to different processes. Then, we use the framework of Clement et al. [60] combined with this primitive to convert a message-passing consensus algorithm that tolerates crash failures into a consensus algorithm that tolerates Byzantine failures. This yields Robust Backup.[2] It uses only static permissions and assumes memories are split into SWMR regions. Therefore, this sub-algorithm works in the traditional shared-memory model with SWMR registers, and it may be of independent interest.

The second sub-algorithm is called *Cheap Quorum*. It uses dynamic permissions to decide in two delays using one signature in common executions. However, the sub-algorithm gives up if the system is not synchronous or there are Byzantine failures.

Finally, we combine both sub-algorithms using ideas from the Abstract framework of Aublin et al. [23]. More precisely, we start by running Cheap Quorum; if it aborts, we run Robust Backup. There is a subtlety: for this idea to work, Robust Backup must decide on a value v if Cheap Quorum decided v previously. To do that, Robust Backup decides on a *preferred value* if at least $f + 1$ processes have this value as input. To do so, we use the classic crash-tolerant Paxos algorithm (run under the Robust Backup algorithm to ensure Byzantine tolerance) but with an initial set-up phase that ensures this safe decision. We call the protocol *Preferential Paxos*.

[2] The attentive reader may wonder why at this point we have not achieved a 2-deciding algorithm already: if we apply Clement et al. [60] to a 2-deciding crash-tolerant algorithm (such as Fast Paxos [35]), will the result not be a 2-deciding Byzantine-tolerant algorithm? The answer is no, because Clement et al. needs reliable broadcast, which incurs at least 6 delays.

2.4.1 The Robust Backup Sub-Algorithm

We develop Robust Backup using the construction by Clement et al. [60], which we now explain. Clement et al. show how to transform a message-passing algorithm $\mathscr{A}$ that tolerates f_P crash failures into a message-passing algorithm that tolerates f_P Byzantine failures in a system where $n \geq 2f_P + 1$ processes, assuming unforgeable signatures and a non-equivocation mechanism. They do so by implementing trusted message-passing primitives, *T-send* and *T-receive*, using non-equivocation and signature verification on every message. Processes include their full history with each message, and then verify locally whether a received message is consistent with the protocol. This restricts Byzantine behavior to crash failures.

To apply this construction in our model, we show that our model can implement non-equivocation and message passing. We first show that shared-memory with SWMR registers (and no memory failures) can implement these primitives, and then show how our model can implement shared-memory with SWMR registers.

2.4.1.1 Reliable Broadcast

Consider a shared-memory system. We present a way to prevent equivocation through a solution to the *reliable broadcast* problem, which we recall below. Note that our definition of reliable broadcast includes the sequence number k (as opposed to being single-shot) so as to facilitate the integration with the Clement et al. construction, as we explain in Section 2.4.1.2.

Definition 2.4.1. Reliable broadcast *is defined in terms of two primitives, broadcast*(k,m) *and deliver*(k,m,q). *When a process* p *invokes broadcast*(k,m) *we say that* p broadcasts (k,m). *When a process* p *invokes deliver*(k,m,q) *we say that* p delivers (k,m) from q. *Each correct process* p *must invoke broadcast*$(k,*)$ *with* k *one higher than* p*'s previous invocation (and first invocation with* k=1*). The following holds:*

1. *If a correct process* p *broadcasts* (k,m)*, then all correct processes eventually deliver* (k,m) *from* p.
2. *If* p *and* q *are correct processes,* p *delivers* (k,m) *from* r*, and* q *delivers* (k,m') *from* r*, then* $m=m'$.
3. *If a correct process delivers* (k,m) *from a correct process* p*, then* p *must have broadcast* (k,m).
4. *If a correct process delivers* (k,m) *from* p*, then all correct processes eventually deliver* (k,m') *from* p *for some* m'.

Algorithm 2.1 shows how to implement reliable broadcast that is tolerant to a minority of Byzantine failures in shared-memory using SWMR registers. To broadcast its k-th message m, p simply signs (k,m) and writes it in slot $Value[p,k,p]$ of its memory[3].

[3]The indexing of the slots is as follows: the first index is the writer of the SWMR register, the second index is the sequence number of the message, and the third index is the sender of the message.

Algorithm 2.1 – Reliable Broadcast.

```
1  Global state:
2  SWMR Value[n,M,n]; initialized to ⊥. Value[p] is array of SWMR(p) registers.
3  SWMR L1Proof[n,M,n]; initialized to ⊥. L1Proof[p] is array of SWMR(p) registers.
4  SWMR L2Proof[n,M,n]; initialized to ⊥. L2Proof[p] is array of SWMR(p) registers.

6  Local state:
7  last[n]: local array with last k delivered from each process. Initially, last[q] = 0
8  state[n]: local array of registers. state[q] ∈ {WaitForSender,WaitForL1Proof,
     ↪ WaitForL2Proof}. Initially, state[q] = WaitForSender

10 Code for process p:
11    broadcast (k,m){
12       Value[p,k,p].write(sign((k,m)));  }

14    for q in Π in parallel {
15       while true { try_deliver(q); }}

17    try_deliver(q) {
18       k = last[q];
19       val = checkL2Proof(q,k);
20       if (val != null) {
21          deliver(k, proof.msg, q);
22          last[q] += 1;
23          state = WaitForSender;
24          return; }
25       if state == WaitForSender {
26          val = Value[q,k,q].read();
27          if (val==⊥ || !sValid(p, val) || key!=k) { return; }
28          Value[p,k,q].write(sign(val));
29          state = WaitForL1Proof; }
30       if state == WaitForL1Proof {
31          checkedVals = ∅;
32          for i ∈Π{
33             val = Value[i,k,q].read();
34             if (val!=⊥ && sValid(p,val) && key==k) { add val to checkedVals; } }
35          if size(checkedVals) ≥ majority and checkedVals contains only one value {
36             l1prf = sign(checkedVals);
37             L1Proof[p,k,q].write(l1prf);
38             state = WaitForL2Proof; } }
39       if state == WaitForL2Proof{
40          checkedL1Proofs = ∅;
41          for i in Π{
42             proof = L1Proof[i,k,q].read();
43             if ( checkL1Proof(proof) ) { add proof to checkedL1Proofs; } }
44          if size(checkedL1Proofs) ≥ majority {
45             l2prf = sign(checkedL1Proofs);
46             L2Proof[p,k,q].write(l2prf); } } }

48    value checkL2proof(q,k) {
49       for i∈Π {
50          proof = L2Proof[i,k,q].read();
51          if (proof != ⊥ && check(proof)) {
52             L2Proof[p,k,q].write(proof);
53             return proof.msg; } }
54       return null; }
```

Delivering a message from another process is a little more involved, requiring verification steps to ensure that all correct processes will eventually deliver the same message and no other. The high-level idea is that before delivering a message (k, m) from q, each process p checks that no other process saw a different value from q, and waits to hear that "enough" other processes also saw the same value. More specifically, each process p has 3 slots per process per sequence number, that only p can write to, but all processes can read from. These slots are initialized to $\perp$, and p uses them to write the values that it has seen. The 3 slots represent 3 levels of 'proofs' that this value is correct; for each process q and sequence number k, p has a slot to write (1) the initial value v it read from q for k, (2) a proof that at least $f+1$ processes saw the same value v from q for k, and (3) a proof that at least $f+1$ processes wrote a proof of seeing value v from q for k in their second slot. We call these slots the Value slot, the L1Proof slot, and the L2Proof slot, respectively.

We note that each such valid proof has signed copies of only one value for the message. Any proof that shows copies of two different values or a value that isn't signed is not considered valid. If a proof has copies of only value v, we say that this proof *supports* v.

To deliver a value v from process q with sequence number k, process p must successfully write a valid proof-of-proofs in its L2Proof slot supporting value v (we call this an L2 proof). It has two options of how to do this; firstly, if it sees a valid L2 proof in some other process i's $L2Proof[i, k, q]$ slot, it copies this proof over to its own L2 proof slot, and can then deliver the value that this proof supports. If p does not find a valid L2 proof in some other process's slot, it must try to construct one itself. We now describe how this is done.

A correct process p goes through three stages when constructing a valid L2 proof for (k, m) from q. In the pseudocode, the three stages are denoted using states that p goes through: WaitForSender, WaitForL1Proof, and WaitForL2Proof.

In the first stage, WaitForSender, p reads q's $Value[q, k, q]$ slot. If p finds a (k, m) pair, p signs and copies it to its $Value[p, k, q]$ slot and enters the WaitForL1Proof state.

In the second stage, WaitForL1Proof, p reads all $Value[i, k, q]$ slots, for $i \in \Pi$. If all the values p reads are correctly signed and equal to (k, m), and if there are at least $f+1$ such values, then p compiles them into an L1 proof, which it signs and writes to $L1Proof[p, k, q]$; p then enters the WaitForL2Proof state.

In the third stage, WaitForL2Proof, p reads all $L1Proof[i, k, q]$ slots, for $i \in \Pi$. If p finds at least $f+1$ valid and signed L1 proofs for (k, m), then p compiles them into an L2 proof, which it signs and writes to $L2Proof[p, k, q]$. The next time that p scans the $L2Proof[\cdot, k, q]$ slots, p will see its own L2 proof (or some other valid proof for (k, m)) and deliver (k, m).

This three-stage validation process ensures the following crucial property: no two valid L2 proofs can support different values. Intuitively, this property is achieved because for both L1 and L2 proofs, at least $f+1$ values of the previous stage must be copied, meaning that at least

one correct process was involved in the quorum needed to construct each proof. Because correct processes read the slots of *all* others at each stage before constructing the next proof, and because they never overwrite or delete values that they already wrote, it is guaranteed that no two correct processes will create valid L1 proofs for different values, since one must see the Value slot of the other. Thus, no process or processes, Byzantine or otherwise, can construct valid L2 proofs for different values.

Notably, a weaker version of broadcast, which does not require Property 4 (i.e., Byzantine consistent broadcast [47, Module 3.11]), can be solved with just the first stage of Algorithm 2.1, without the L1 and L2 proofs. The purpose of those proofs is to ensure the 4th property holds; that is, to enable all correct processes to deliver a value once some correct process delivered.

In Appendix A.1, we formally prove the above intuition and arrive at the following lemma.

Lemma 2.4.2. *Reliable broadcast can be solved in shared-memory with SWMR regular registers with $n \geq 2f + 1$ processes.*

2.4.1.2 Applying Clement et al.'s Construction

Clement et al. show that given unforgeable transferable signatures and non-equivocation, one can reduce Byzantine failures to crash failures in message passing systems [60]. They define non-equivocation as a predicate $valid_p$ for each process p, which takes a sequence number and a value and evaluates to true for just one value per sequence number. All processes must be able to call the same $valid_p$ predicate, which always terminates every time it is called.

We now show how to use reliable broadcast to implement messages with transferable signatures and non-equivocation as defined by Clement et al. [60]. Note that our reliable broadcast mechanism already involves the use of transferable signatures, so to send and receive signed messages, one can simply use broadcast and deliver those messages. However, simply using broadcast and deliver is not quite enough to satisfy the requirements of the $valid_p$ predicate of Clement et al. The problem occurs when trying to validate nested messages recursively.

In particular, recall that in Clement et al's construction, whenever a message is sent, the entire history of that process, including all messages it has sent and received, is attached. Consider two Byzantine processes q_1 and q_2, and assume that q_1 attempts to equivocate in its kth message, signing both (k, m) and (k, m'). Assume therefore that no correct process delivers any message from q_1 in its kth round. However, since q_2 is also Byzantine, it could claim to have delivered (k, m) from q_1. If q_2 then sends a message that includes (q, k, m) as part of its history, a correct process p receiving q_2's message must recursively verify the history q_2 sent. To do so, p can call `try_deliver` on (q_1, k). However, since no correct process delivered any message from (q_1, k), it is possible that this call never returns.

To solve this issue, we introduce a `validate` operation that can be used along with broadcast and deliver to validate the correctness of a given message. The `validate` operation is very

Algorithm 2.2 – Validate Operation for Reliable Broadcast.

```
1 bool validate(q,k,m){
2     val = checkL2proof(q,k);
3     if (val == m) {
4         return true; }
5     return false; }
```

simple: it takes in a process id, a sequence number, and a message value m, and simply runs the `checkL2proof` helper function. If the function returns a proof supporting m, `validate` returns true. Otherwise it returns false. The pseudocode is shown in Algorithm 2.2.

In this way, Algorithms 2.1 and 2.2 together provide signed messages and a non-equivocation primitive. Thus, combined with the construction of Clement et al. [60], we immediately get the following result.

Theorem 2.4.3. *There exists an algorithm for weak Byzantine agreement in a shared-memory system with SWMR regular registers, signatures, and up to f_P process crashes where $n \geq 2f_P + 1$.*

Non-equivocation in our model. To convert the above algorithm to our model, where memory may fail, we use the ideas in [3, 20, 128] to implement failure-free SWMR regular registers from the fail-prone memory, and then run weak Byzantine agreement using those regular registers. To implement an SWMR register, a process writes or reads all memories, and waits for a majority to respond. When reading, if p sees exactly one distinct non-$\bot$ value v across the memories, it returns v; otherwise, it returns $\bot$.

Definition 2.4.4. *Let $\mathscr{A}$ be a message-passing algorithm.* Robust Backup($\mathscr{A}$) *is the algorithm $\mathscr{A}$ in which all $send$ and $receive$ operations are replaced by T-send and T-receive operations (respectively) implemented with reliable broadcast.*

Thus we get the following lemma, from the result of Clement et al. [60], Lemma 2.4.2, and the above handling of memory failures.

Lemma 2.4.5. *If $\mathscr{A}$ is a consensus algorithm that is tolerant to f process crash failures, then Robust Backup($\mathscr{A}$) is a weak Byzantine agreement algorithm that is tolerant to up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$ in the message-and-memory model.*

The following theorem is an immediate corrolary of the lemma.

Theorem 2.4.6. *There exists an algorithm for Weak Byzantine Agreement in a message-and-memory model with up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$.*

2.4.2 The Cheap Quorum Sub-Algorithm

We now give an algorithm that decides in two delays in common executions in which the system is synchronous and there are no failures. It requires only one signature for a fast decision, whereas the best prior algorithm requires $6f_P + 2$ signatures and $n \geq 3f_P + 1$ [23]. Our algorithm, called Cheap Quorum, is not in itself a complete consensus algorithm; it may abort in some executions. If Cheap Quorum aborts, it outputs an *abort value*, which is used to initialize the Robust Backup so that their composition preserves weak Byzantine agreement. This composition is inspired by the Abstract framework of Aublin et al. [23].

The algorithm has a special process ℓ, say $\ell = p_1$, which serves both as a *leader* and a *follower*. Other processes act only as *followers*. The memory is partitioned into $n+1$ regions denoted $Region[p]$ for each $p \in \Pi$, plus an extra one for p_1, $Region[\ell]$ in which it proposes a value. Initially, $Region[p]$ is a regular SWMR region where p is the writer. Unlike in Algorithm 2.1, some of the permissions are dynamic; processes may remove p_1's write permission to $Region[\ell]$ (i.e., the *legalChange* function returns false to any permission change requests, except for ones revoking p_1's permission to write on $Region[\ell]$).

Processes initially execute under a *normal* mode in common-case executions, but may switch to *panic* mode if they intend to abort, as in [23]. The pseudo-code of the normal mode is in Algorithm 2.3. $Region[p]$ contains three registers $Value[p]$, $Panic[p]$, $Proof[p]$ initially set to $\perp$, *false*, $\perp$. To propose v, the leader p_1 signs v and writes it to $Value[\ell]$. If the write is successful (it may fail because its write permission was removed), then p_1 decides v; otherwise p_1 calls *Panic_mode*(). Note that all processes, including p_1, continue their execution after deciding. However, p_1 never decides again if it decided as the leader. A follower q checks if p_1 wrote to $Value[\ell]$ and, if so, whether the value is properly signed. If so, q signs v, writes it to $Value[q]$, and waits for other processes to write the same value to $Value[*]$. If q sees $2f+1$ copies of v signed by different processes, q assembles these copies in a *unanimity proof*, which it signs and writes to $Proof[q]$. q then waits for $2f+1$ unanimity proofs for v to appear in $Proof[*]$, and checks that they are valid, in which case q decides v. This waiting continues until a timeout expires[4], at which time q calls *Panic_mode*(). In *Panic_mode*() (shown in Algorithm 2.4), a process p sets $Panic[p]$ to *true* to tell other processes it is panicking; other processes periodically check to see if they should panic too. p then removes write permission from $Region[\ell]$, and decides on a value to abort: either $Value[p]$ if it is non-$\perp$, $Value[\ell]$ if it is non-$\perp$, or p's input value. If p has a unanimity proof in $Proof[p]$, it adds it to the abort value.

In Appendix A.2, we prove the correctness of Cheap Quorum, and in particular we show the following two important agreement properties:

Lemma 2.4.7 (Cheap Quorum Decision Agreement)**.** *Let p and q be correct processes. If p decides v_1 while q decides v_2, then $v_1 = v_2$.*

Lemma 2.4.8 (Cheap Quorum Abort Agreement)**.** *Let p and q be correct processes (possibly*

[4]The timeout is chosen to be an upper bound on the communication, processing and computation delays in the common case.

Algorithm 2.3 – Cheap Quorum normal operation—code for process p.

```
1  Leader code
2  propose(v) {
3      sign(v);
4      status = Value[ℓ].write(v);
5      if (status == nak) Panic_mode();
6      else decide(v); }

8  Follower code
9  propose(w){
10     do {v = read(Value[ℓ]);
11         for all q ∈ Π do pan[q] = read(Panic[q]);
12     } until (v ≠ ⊥ || pan[q] == true for some q || timeout);
13     if (v ≠ ⊥ && sValid(p1,v)) {
14         sign(v);
15         write(Value[p],v);
16         do {for all q ∈ Π do val[q] = read(Value[q]);
17             if |{q : val[q] == v}| ≥ n then {
18                 Proof[p].write(sign(val[1..n]));
19                 for all q ∈ Π do prf[q] = read(Proof[q]);
20                 if |{q : verifyProof(prf[q]) == true}| ≥ n { decide(v); exit; } }
21             for all q ∈ Π do pan[q] = read(Panic[q]);
22         } until (pan[q] == true for some q || timeout); }
23     Panic_mode();}
```

identical). If p decides v in Cheap Quorum while q aborts from Cheap Quorum, then v will be q's abort value. Furthermore, if p is a follower, q's abort proof is a correct unanimity proof.

The above construction assumes a fail-free memory with regular registers, but we can extend it to tolerate memory failures using the approach of Section 2.4.1, noting that each register has a single writer process.

Algorithm 2.4 – Cheap Quorum panic mode—code for process p.

```
1  panic_mode(){
2      Panic[p] = true;
3      changePermission(Region[ℓ], R: Π, W: {}, RW: {});  // remove write
         ↪ permission
4      v = read(Value[p]);
5      prf = read(Proof[p]);
6      if (v ≠⊥){  Abort with <v, prf>; return; }
7      LVal = read(Value[ℓ]);
8      if (LVal ≠⊥) {Abort with <LVal, ⊥>; return;}
9      Abort with <myInput, ⊥>; }
```

2.4.3 Putting it Together: the Fast & Robust Algorithm

The final algorithm, called Fast & Robust, combines Cheap Quorum (§2.4.2) and Robust Backup (§2.4.1), as we now explain. Recall that Robust Backup is parameterized by a message-passing consensus algorithm $\mathscr{A}$ that tolerates crash-failures. $\mathscr{A}$ can be any such algorithm (e.g., Paxos).

Roughly, in Fast & Robust, we run Cheap Quorum and, if it aborts, we use a process's abort value as its input value to Robust Backup. However, we must carefully glue the two algorithms together to ensure that if some correct process decided v in Cheap Quorum, then v is the only value that can be decided in Robust Backup.

For this purpose, we propose a simple wrapper for Robust Backup, called *Preferential Paxos* (Algorithm 2.5). Preferential Paxos first runs a set-up phase, in which processes may adopt new values, and then runs Robust Backup with the new values. More specifically, there are some *preferred* input values $v_1 \dots v_k$, ordered by priority. We guarantee that every process adopts one of the top $f+1$ priority inputs. In particular, this means that if a majority of processes get the highest priority value, v_1, as input, then v_1 is guaranteed to be the decision value. The set-up phase is simple; all processes send each other their input values. Each process p waits to receive $n-f$ such messages, and adopts the value with the highest priority that it sees. This is the value that p uses as its input to Paxos.

Algorithm 2.5 – Preferential Paxos—code for process p.

```
1  propose((v, priorityTag)){
2      T-send(v, priorityTag) to all;
3      Wait to T-receive (val,priorityTag) from n − f_P processes;
4      best = value with highest priority out of messages received;
5      RobustBackup(Paxos).propose(best); }
```

Lemma 2.4.9 (Preferential Paxos Priority Decision)**.** *Preferential Paxos implements weak Byzantine agreement with $n \geq 2f_P + 1$ processes. Furthermore, let $v_1, \dots, v_n$ be the input values of an instance C of Preferential Paxos, ordered by priority. The decision value of correct processes is always one of $v_1, \dots, v_{f+1}$.*

Proof. Recall that *T-send* and *T-receive* are the trusted message passing primitives that are implemented in [60] using non-equivocation and signatures.

By Lemma 2.4.5, Robust Backup(Paxos) solves weak Byzantine agreement with $n \geq 2f_P + 1$ processes. Note that before calling Robust Backup(Paxos), each process may change its input, but only to the input of another process. Thus, by the correctness and fault tolerance of Paxos, Preferential Paxos clearly solves weak Byzantine agreement with $n \geq 2f_P + 1$ processes. Thus we only need to show that Preferential Paxos satisfies the priority decision property with $2f_P + 1$ processes that may only fail by crashing.

Since Robust Backup(Paxos) satisfies validity, if all processes call Robust Backup(Paxos) in

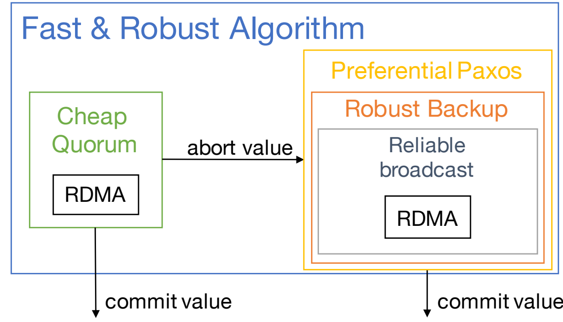


Figure 2.2 – Interactions of the components of the Fast & Robust Algorithm.

line 5 with a value v that is one of the $f_P + 1$ top priority values (that is, $v \in \{v_1, \ldots, v_{f_P+1}\}$), then the decision of correct processes will also be in $\{v_1, \ldots, v_{f_P+1}\}$. So we just need to show that every process indeed adopts one of the top $f_P + 1$ values. Note that each process p waits to see $n - f_P$ values, and then picks the highest priority value that it saw. No process can lie or pick a different value, since we use T-send and T-receive throughout. Thus, p can miss at most f_P values that are higher priority than the one that it adopts. □

We can now describe Fast & Robust in detail. We start executing Cheap Quorum. If Cheap Quorum aborts, we execute Preferential Paxos, with each process receiving its abort value from Cheap Quorum as its input value to Preferential Paxos. We define the priorities of inputs to Preferential Paxos as follows.

Definition 2.4.10 (Input Priorities for Preferential Paxos)**.** *The input values for Preferential Paxos as it is used in Fast & Robust are split into three sets (here, p_1 is the leader of Cheap Quorum):*

- $T = \{v \mid v \text{ contains a correct unanimity proof}\}$
- $M = \{v \mid v \notin T \wedge v \text{ contains the signature of } p_1\}$
- $B = \{v \mid v \notin T \wedge v \notin M\}$

The priority order of the input values is such that for all values $v_T \in T$, $v_M \in M$, and $v_B \in B$, $priority(v_T) > priority(v_M) > priority(v_B)$.

Figure 2.2 shows how the various algorithms presented in this section come together to form the Fast & Robust algorithm.

In Appendices A.2 and A.3, we show that Fast & Robust is correct, with the following key lemma:

Lemma 2.4.11 (Composition Lemma)**.** *If some correct process decides a value v in Cheap Quorum before an abort, then v is the only value that can be decided in Preferential Paxos with priorities as defined in Definition 2.4.10.*

Theorem 2.4.12. *There exists a 2-deciding algorithm for Weak Byzantine Agreement in a message-and-memory model with up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$.*

2.5 Crash Failures

We now restrict ourselves to crash failures of processes and memories. Clearly, we can use the algorithms of Section 2.4 in this setting, to obtain a 2-deciding consensus algorithm with $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$. However, this is overkill since those algorithms use sophisticated mechanisms (signatures, non-equivocation) to guard against Byzantine behavior. With only crash failures, we now show it is possible to retain the efficiency of a 2-deciding algorithm while improving resiliency. We give a 2-deciding algorithm that allows the crash of all but one process ($n \geq f_P + 1$) and a minority of memories ($m \geq 2f_M + 1$).

Our starting point is the Disk Paxos algorithm [92], which works in a system with processes and memories where $n \geq f_P + 1$ and $m \geq 2f_M + 1$. This is our resiliency goal, but Disk Paxos takes four delays in common executions. Our new algorithm, called Protected Memory Paxos, removes two delays; it retains the structure of Disk Paxos but uses permissions to skip steps. Initially some fixed leader $\ell = p_1$ has exclusive write permission to all memories; if another process becomes leader, it takes the exclusive permission. Having exclusive permission permits a leader ℓ to optimize execution, because ℓ can do two things simultaneously: (1) write its consensus proposal and (2) determine whether another leader took over. Specifically, if ℓ succeeds in (1), it knows no leader ℓ' took over because ℓ' would have taken the permission. Thus ℓ avoids the last read in Disk Paxos, saving two delays. Of course, care must be taken to implement this without violating safety.

The pseudocode of Protected Memory Paxos is in Algorithm 2.6. Each memory has one memory region, and at any time exactly one process can write to the region. Each memory i holds a register $slot[i, p]$ for each process p. Intuitively, $slot[i, p]$ is intended for p to write, but p may not have write permission to do that if it is not the leader—in that case, no process writes $slot[i, p]$.

When a process p becomes the leader, it must execute a sequence of steps on a majority of the memories to successfully commit a value. It is important that p execute *all* of these steps on each of the memories that counts toward its majority; otherwise two leaders could miss each other's values and commit conflicting values. We therefore present the pseudocode for this algorithm in a *parallel-for* loop (lines 18–37), with one thread per memory that p accesses. The algorithm has two phases similar to Paxos, where the second phase may only begin after the first phase has been completed for a majority of the memories. We represent this in the code with a barrier that waits for a majority of the threads.

When a process p becomes leader, it executes the prepare phase (the first leader p_1 can skip this phase in its first execution of the loop), where, for each memory, p attempts to (1) acquire

Algorithm 2.6 – Protected Memory Paxos—code for process p.

```
1  Registers: for i=1..m, p ∈Π,
2      slot[i,p]: tuple (minProp, accProp, value)// in memory i
3  Ω: failure detector that returns current leader

5  startPhase2(i) {
6      add i to ListOfReady processes;
7      while (size(ListOfReady)<majority of memories) {}
8      Phase2Started = true; }

10 propose(v) {
11 repeat forever {
12     wait until Ω == p; // wait to become leader
13     propNr = a higher value than any proposal number seen before;
14     CurrentVal = v;
15     CurrentMaxProp = 0;
16     Phase2Started = false;
17     ListOfReady = ∅;
18     for every memory i in parallel {
19         if (p != p1 || not first attempt) {
20             getPermission(i);
21             success = write(slot[i,p], (propNr,⊥,⊥));
22             if (not success) { abort(); }
23             vals = read all slots from i;
24             if (vals contains a non-null value) {
25                 val = v ∈ vals with highest propNr;
26                 if (val.propNr>propNr) { abort(); }
27                 atomic {
28                     if(val.propNr>CurrentMaxProp){
29                          if (Phase2Started) {abort();}
30                          CurrentVal = val.value;
31                          CurrentMaxProp = val.propNr;
32                    } } }
33             startPhase2(i);}
34         // done phase 1 or (p == p1 && p1's first attempt)
35         success = write(slot[i,p], (propNr,propNr,CurrentVal));
36         if (not success) { abort(); }
37     } until this has been done at a majority of the memories, or until '
         ↪ abort' has been called
38     if (loop completed without abort) {
39         decide CurrentVal; } } }
```

exclusive write permission, (2) write a new proposal number in its slot, and (3) read all slots of that memory. p waits to succeed in executing these steps on a majority of the memories. If any of p's writes fail or p finds a proposal with a higher proposal number, then p gives up. This is represented with an `abort` in the pseudocode; when an `abort` is executed, the for loop terminates. We assume that when the for loop terminates—either because some thread has aborted or because a majority of threads have reached the end of the loop—all threads of the for loop are terminated and control returns to the main loop (lines 11–37).

If p does not abort, it adopts the value with highest proposal number of all those it read in

the memories. To make it clear that races should be avoided among parallel threads in the pseudocode, we wrap this part in an `atomic` environment.

In the next phase, each of p's threads writes its value to its slot on its memory. If a write fails, p gives up. If p succeeds, this is where we optimize time: p can simply decide, whereas Disk Paxos must read the memories again.

Note that it is possible that some of the memories that made up the majority that passed the initial barrier may crash later on. To prevent p from stalling forever in such a situation, it is important that straggler threads that complete phase 1 later on be allowed to participate in phase 2. However, if such a straggler thread observes a more up-to-date value in its memory than the one adopted by p for phase 2, this must be taken into account. In this case, to avoid inconsistencies, p must abort its current attempt and restart the loop from scratch.

The code ensures that some correct process eventually decides, but it is easy to extend it so all correct processes decide [54], by having a decided process broadcast its decision. Also, the code shows one instance of consensus, with p_1 as initial leader. With many consensus instances, the leader terminates one instance and becomes the default leader in the next.

Theorem 2.5.1. *Consider a message-and-memory model with up to f_P process crashes and f_M memory crashes, where $n \geq f_P + 1$, $m \geq 2f_M + 1$. There exists a 2-deciding consensus algorithm.*

2.6 Dynamic Permissions are Necessary for Efficient Consensus

In §2.5, we showed how dynamic permissions can improve the performance of Disk Paxos. Are dynamic permissions necessary? We prove that with shared memory (or disks) alone, one cannot achieve 2-deciding consensus, even if the memory never fails, it has static permissions, processes may only fail by crashing, and the system is partially synchronous in the sense that eventually there is a known upper bound on the time it takes a correct process to take a step [87]. This result applies a fortiori to the Disk Paxos model [92].

Theorem 2.6.1. *Consider a partially synchronous shared-memory model with registers, where registers can have arbitrary static permissions, memory never fails, and at most one processes may fail by crashing. No consensus algorithm is 2-deciding.*

Proof. Assume by contradiction that A is an algorithm in the stated model that is 2-deciding. That is, there is some execution E of A in which some process p decides a value v with 2 delays. We denote by R and W the set of objects which p reads and writes in E respectively. Note that since p decides in 2 delays in E, R and W must be disjoint, by the definition of operation delay and the fact that a process has at most one outstanding operation per object. Furthermore, p must issue all of its read and writes without waiting for the response of any operation.

Consider an execution E' in which p reads from the same set R of objects and writes the same values as in E to the same set W of objects. All of the read operations that p issues return

by some time t_0, but the write operations of p are delayed for a long time. Another process p' begins its proposal of a value $v' \neq v$ after t_0. Since no process other than p' writes to any objects, E' is indistinguishable to p' from an execution in which it runs alone. Since A is a correct consensus algorithm that terminates if there is no contention, p' must eventually decide value v'. Let t' be the time at which p' decides. All of p's write operations terminate and are linearized in E' after time t'. Execution E' is indistinguishable to p from execution E, in which it ran alone. Therefore, p decides $v \neq v'$, violating agreement. □

Theorem 2.6.1, together with the Fast Paxos algorithm of Lamport [149], shows that an atomic read-write shared memory model is strictly weaker than the message passing model in its ability to solve consensus quickly. This result may be of independent interest, since often the classic shared memory and message passing models are seen as equivalent, because of the seminal computational equivalence result of Attiya, Bar-Noy, and Dolev [20]. Interestingly, it is known that shared memory can tolerante more failures when solving consensus (with randomization or partial synchrony) [19, 39], and therefore it seems that perhaps shared memory is strictly stronger than message passing for solving consensus. However, our result shows that there are aspects in which message passing is stronger than shared memory. In particular, message passing can solve consensus faster than shared memory in well-behaved executions.

2.7 RDMA in Practice

Our model is meant to reflect capabilities of RDMA, while providing a clean abstraction to reason about. We now give an overview of how RDMA works, and how features of our model can be implemented using RDMA.

RDMA enables a remote process to access local memory directly through the network interface card (NIC), without involving the CPU. For a piece of local memory to be accessible to a remote process p, the CPU has to *register* that memory region and associate it with the appropriate connection (called *Queue Pair*) for p. The association of a registered memory region and a queue pair is done indirectly through a *protection domain*: both memory regions and queue pairs are associated with a protection domain, and a queue pair q can be used to access a memory region r if q and r and in the same protection domain. The CPU must also specify what access level (read, write, read-write) is allowed to the memory region in each protection domain. A local memory area can thus be registered and associated with several queue pairs, with the same or different access levels, by associating it with one or more protection domains. Each RDMA connection can be used by the remote server to access registered memory regions using a unique region-specific key created as a part of the registration process.

As highlighted by previous work [199], failures of the CPU, NIC and DRAM can be seen as independent (e.g., arbitrary delays, too many bit errors, failed ECC checks, respectively). For instance, *zombie servers* in which the CPU is blocked but RDMA requests can still be served

account for roughly half of all failures [199]. This motivates our choice to treat processes and memory separately in our model. In practice, if a CPU fails permanently, the memory will also become unreachable through RDMA eventually; however, in such cases memory may remain available long enough for ongoing operations to complete. Also, in practical settings it is possible for full-system crashes to occur (e.g., machine restarts), which correspond to a process and a memory failing at the same time—this is allowed by our model.

Memory regions in our model correspond to RDMA memory regions. Static permissions can be implemented by making the appropriate memory region registration before the execution of the algorithm; these permissions then persist during execution without CPU involvement. Dynamic permissions require the host CPU to change the access levels; this should be done in the OS kernel: the kernel creates regions and controls their permissions, and then shares memory with user-space processes. In this way, Byzantine processes cannot change permissions illegally. The assumption is that the kernel is not Byzantine. Alternatively, future hardware support similar to SGX could even allow parts of the kernel to be Byzantine.

Using RDMA, a process p can grant permissions to a remote process q by registering memory regions with the appropriate access permissions (read, write, or read/write) and sending the corresponding key to q. p can revoke permissions dynamically by simply deregistering the memory region. In practice, changing RDMA permissions can be several orders of magnitude slower than remote reads or write. In this chapter, we ignore this cost, since we are interested in the common-case complexity of algorithms (and permission changes do not occur in the common case). We examine this cost in more depth in Chapter 3, when we consider ways to improve failover performance.

For our reliable broadcast algorithm, each process can register the two dimensional array of values in read-only mode with a protection domain. All the queue pairs used by that process are also created in the context of the same protection domain. Additionally, the process can preserve write access permission to its row via another registration of just that row with the protection domain, thus enabling single-writer multiple-reader access. Thereafter the reliable broadcast algorithm can be implemented trivially by using RDMA reads and writes by all processes. Reliable broadcast with unreliable memories is similarly straightforward since failure of the memory ensures that no process will be able to access the memory.

For Cheap Quorum, the static memory region registrations are straightforward as above. To revoke the leader's write permission, it suffices for a region's host process to deregister the memory region. Panic messages can be relayed using RDMA message sends.

In our crash-only consensus algorithm, we leverage the capability of registering overlapping memory regions in a protection domain. As in above algorithms, each process uses one protection domain for RDMA accesses. Queue pairs for connections to all other processes are associated with this protection domain. The process' entire slot array is registered with the protection domain in read-only mode. In addition, the same slot array can be dynamically registered (and deregistered) in write mode based on incoming write permission requests: A

proposer requests write permission using an RDMA message send. In response, the acceptor first deregisters write permission for the immediate previous proposer. The acceptor thereafter registers the slot array in write mode and responds to the proposer with the new key associated with the newly registered slot array. Reads of the slot array are performed by the proposer using RDMA reads. Subsequent second phase RDMA write of the value can be performed on the slot array as long as the proposer continues to have write permission to the slot array. The RDMA write fails if the acceptor granted write permission to another proposer in the meantime.

3 Microsecond Consensus for Microsecond Applications

In the previous chapter, we showed that in a model which captures the properties of RDMA, consensus can be solved in a single round trip, with better fault-tolerance than is possible in the traditional message-passing or shared-memory models. In this chapter, we focus on crash-faults and look at the practical applicability of the results from the previous chapter. Namely, we consider the problem of making apps fault-tolerant through replication, when apps operate at the microsecond scale, as in finance, embedded computing, and microservices apps. These apps need a replication scheme that also operates at the microsecond scale, otherwise replication becomes a burden. We propose Mu, a system that takes less than 1.3 microseconds to replicate a (small) request in memory, and less than a millisecond to fail-over the system—this cuts the replication and fail-over latencies of the prior systems by at least 61% and 90%. Mu implements bona fide state machine replication/consensus (SMR) with strong consistency for a generic app, but it really shines on microsecond apps, where even the smallest overhead is significant. To provide this performance, Mu introduces a new SMR protocol that carefully leverages RDMA. Roughly, in Mu a leader replicates a request by simply writing it directly to the log of other replicas using RDMA, without any additional communication. Doing so, however, introduces the challenge of handling concurrent leaders, changing leaders, garbage collecting the logs, and more—challenges that we address in this chapter through a judicious combination of RDMA permissions and distributed algorithmic design. We implemented Mu and used it to replicate several systems: a financial exchange app called Liquibook, Redis, Memcached, and HERD [132]. Our evaluation shows that Mu incurs a small replication latency, in some cases being the only viable replication system that incurs an acceptable overhead.

3.1 Introduction

Enabled by modern technologies such as RDMA, Microsecond-scale computing is emerging as a must [26]. A microsecond app might be expected to process a request in 10 microseconds. Areas where software systems care about microsecond performance include finance (e.g., trading systems), embedded computing (e.g., control systems), and microservices (e.g., key-

value stores). Some of these areas are critical and it is desirable to replicate their microsecond apps across many hosts to provide high availability, due to economic, safety, or robustness reasons. Typically, a system may have hundreds of microservice apps [93], some of which are stateful and can disrupt a global execution if they fail (e.g., key-value stores)—these apps should be replicated for the sake of the whole system.

The golden standard to replicate an app is State Machine Replication (SMR) [211], whereby replicas execute requests in the same total order determined by a consensus protocol. Unfortunately, traditional SMR systems add hundreds of microseconds of overhead even on a fast network [119]. Recent work explores modern hardware in order to improve the performance of replication [129, 131, 136, 140, 199, 225]. The fastest of these (e.g., Hermes [136], DARE [199], and HovercRaft [140]) induce however an overhead of several microseconds, which is clearly high for apps that themselves take few microseconds. Furthermore, when a failure occurs, prior systems incur a prohibitively large fail-over time in the tens of *milli*seconds (not microseconds). For instance, HovercRaft takes 10 milliseconds, DARE 30 milliseconds, and Hermes at least 150 milliseconds. The rationale for such large latencies are timeouts that account for the natural fluctuations in the latency of modern networks. Improving replication and fail-over latencies requires fundamentally new techniques.

We propose Mu, a new SMR system that adds less than 1.3 microseconds to replicate a (small) app request, with the 99th-percentile at 1.6 microseconds. Although Mu is a general-purpose SMR scheme for a generic app, Mu really shines with microsecond apps, where even the smallest replication overhead is significant. Compared to the fastest prior system, Mu is able to cut 61% of its latency. This is the smallest latency possible with current RDMA hardware, as it corresponds to one round of *one-sided* communication.

To achieve this performance, Mu introduces a new SMR protocol that carefully leverages RDMA for replication. Our protocol reaches consensus and replicates a request with just one round of parallel RDMA write operations on a majority of replicas. This is in contrast to prior systems, which take multiple rounds [129, 199, 225] or resort to two-sided communication [119, 131, 142, 169]. Roughly, in Mu the leader replicates a request by simply using RDMA to write it to the log of each replica, without additional rounds of communication. Doing this correctly is challenging because concurrent leaders may try to write to the logs simultaneously. In fact, the hardest part of most replication protocols is the mechanism to protect against races of concurrent leaders (e.g., Paxos proposal numbers [146]). Traditional replication implements this mechanism using send-receive communication (two-sided operations) or multiple rounds of communication. Instead, Mu uses RDMA write permissions to guarantee that a replica's log can be written by only one leader. Critical to correctness are the mechanisms to change leaders and garbage collect logs, as we describe in the chapter.

Mu also improves fail-over time to just 873 microseconds, with the 99-th percentile at 945 microseconds, which cuts fail-over time of prior systems by an order of magnitude. The fact that Mu significantly improves both replication overhead and fail-over latency is perhaps

surprising: folklore suggests a trade-off between the latencies of replication in the fast path, and fail-over in the slow path.

The fail-over time of Mu has two parts: failure detection and leader change. For failure detection, traditional SMR systems typically use a timeout on heartbeat messages from the leader. Due to large variances in network latencies, timeout values are in the 10–100ms even with the fastest networks. This is clearly high for microsecond apps. Mu uses a conceptually different method based on a pull-score mechanism over RDMA. The leader increments a heartbeat counter in its local memory, while other replicas use RDMA to periodically read the counter and calculate a badness score. The score is the number of successive reads that returned the same value. Replicas declare a failure if the score is above a threshold, corresponding to a timeout. Different from the traditional heartbeats, this method can use an aggressively small timeout without false positives because network delays slow down the reads rather than the heartbeat. In this way, Mu detects failures usually within ~600 microseconds. This is bottlenecked by variances in process scheduling, as we discuss later.

For leader change, the latency comes from the cost of changing RDMA write permissions, which with current NICs are hundreds of microseconds. This is higher than we expected: it is far slower than RDMA reads and writes, which go over the network. We attribute this delay to a lack of hardware optimization. RDMA has many methods to change permissions: (1) re-register memory regions, (2) change queue-pair access flags, or (3) close and reopen queue pairs. We carefully evaluate the speed of each method and propose a scheme that combines two of them using a fast-slow path to minimize latency. Despite our efforts, the best way to cut this latency further is to improve the NIC hardware.

We prove that Mu provides strong consistency in the form of linearizability [116], despite crashes and asynchrony, and it ensures liveness under the same assumptions as Paxos [146].

We implemented Mu and used it to replicate several apps: a financial exchange app called Liquibook [161], Redis, Memcached, and an RDMA-based key-value stored called HERD [132].

We evaluate Mu extensively, by studying its replication latency stand-alone or integrated into each of the above apps. We find that, for some of these apps (Liquibook, HERD), Mu is the only viable replication system that incurs a reasonable overhead. This is because Mu's latency is significantly lower by a factor of at least 2.7× compared to other replication systems. We also report on our study of Mu's fail-over latency, with a breakdown of its components, suggesting ways to improve the infrastructure to further reduce the latency.

Mu has some limitations. First, Mu relies on RDMA and so it is suitable only for networks with RDMA, such as local area networks, but not across the wide area. Second, Mu is an in-memory system that does not persist data in stable storage—doing so would add additional latency dependent on the device speed [1]. However, we observe that the industry is working on extensions of RDMA for persistent memory, whereby RDMA writes can be flushed at a remote

[1] For fairness, all SMR systems that we compare against also operate in-memory.

persistent memory with minimum latency [216]—once available, this extension will provide persistence for Mu.

To summarize, we make the following contributions:

- We propose Mu, a new SMR system with low replication and fail-over latencies.
- To achieve its performance, Mu leverages RDMA permissions and a scoring mechanism over heartbeat counters.
- We give the complete correctness proof of Mu (see Appendix B).
- We implement Mu, and evaluate both its raw performance and its performance in microsecond apps. Results show that Mu significantly reduces replication latencies to an acceptable level for microsecond apps.

One might argue that Mu is ahead of its time, as most apps today are not yet microsecond apps. However, this situation is changing. We already have important microsecond apps in areas such as trading, and more will come as existing timing requirements become stricter and new systems emerge as the composition of a large number of microservices (§3.2.1).

3.2 Background

3.2.1 Microsecond Applications and Computing

Apps that are consumed by humans typically work at the millisecond scale: to the human brain, the lowest reported perceptible latency is 13 milliseconds [201]. Yet, we see the emergence of apps that are consumed not by humans but by other computing systems. An increasing number of such systems must operate at the microsecond scale, for competitive, physical, or composition reasons. Schneider [210] speaks of a microsecond market where traders spend massive resources to gain a microsecond advantage in their high-frequency trading. Industrial robots must orchestrate their motors with microsecond granularity for precise movements [15]. Modern distributed systems are composed of hundreds [93] of stateless and stateful microservices, such as key-value stores, web servers, load balancers, and ad services—each operating as an independent app whose latency requirements are gradually decreasing to the microsecond level [36], as the number of composed services is increasing. With this trend, we already see the emergence of key-value stores with microsecond latency (e.g., [131, 186]).

To operate at the microsecond scale, the computing ecosystem must be improved at many layers. This is happening gradually by various recent efforts. Barroso et al. [26] argue for better support of microsecond-scale events. The latest Precision Time Protocol improves clock synchronization to achieve submicrosecond accuracy [13]. And other recent work improves

CPU scheduling [36, 190, 202], thread management [204], power management [203], RPC handling [70, 131], and the network stack [190]—all at the microsecond scale. Mu fits in this context, by providing microsecond SMR.

3.2.2 State Machine Replication

State Machine Replication (SMR) replicates a service (e.g., a key-value storage system) across multiple physical servers called *replicas*, such that the system remains available and consistent even if some servers fail. SMR provides strong consistency in the form of linearizability [116]. A common way to implement SMR, which we adopt in this chapter, is as follows: each replica has a copy of the service software and a log. The log stores client requests. We consider non-durable SMR systems [125, 130, 160, 164, 192, 208], which keep state in memory only, without logging updates to stable storage. Such systems make an item of data reliable by keeping copies of it in the memory of several nodes. Thus, the data remains recoverable as long as there are fewer simultaneous node failures than data copies [199].

A consensus protocol ensures that all replicas agree on what request is stored in each slot of the log. Replicas then apply the requests in the log (i.e., execute the corresponding operations), in log order. Assuming that the service is deterministic, this ensures all replicas remain in sync. We adopt a leader-based approach, in which a dynamically elected replica called the *leader* communicates with the clients and sends back responses after requests reach a majority of replicas. We assume a *crash-failure* model: servers may fail by crashing, after which they stop executing.

Recall that a consensus protocol must ensure *safety* and *liveness* properties. Safety here means (1) *agreement* (different replicas do not obtain different values for a given log slot) and (2) *validity* (replicas do not obtain spurious values). Liveness means *termination*—every live replica eventually obtains a value. We guarantee agreement and validity in an asynchronous system, while termination requires eventual synchrony and a majority of non-crashed replicas, as in typical consensus protocols.

3.2.3 RDMA

Recall from Chapter 2 that Remote Direct Memory Access (RDMA) allows a host to access the memory of another host without involving the processor at the other host. RDMA enables low-latency communication by bypassing the OS kernel and by implementing several layers of the network stack in hardware.

RDMA supports many operations: Send/Receive, Write/Read, and Atomics (compare-and-swap, fetch-and-increment). Because of their lower latency, we use only RDMA Writes and Reads. RDMA has several transports; we use Reliable Connection (RC) to provide in-order reliable delivery.

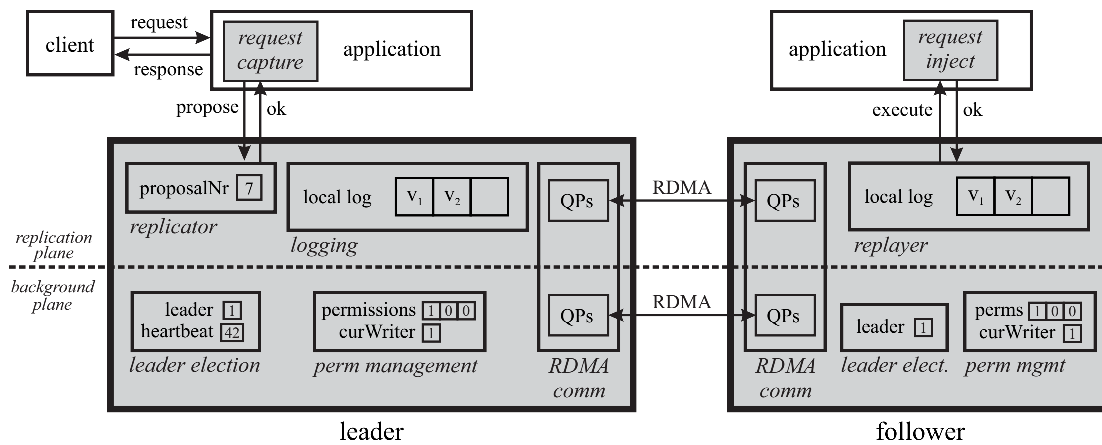


Figure 3.1 – Architecture of Mu. Grey color shows Mu components. A replica is either a leader or a follower, with different behaviors. The leader captures client requests and writes them to the local logs of all replicas. Followers replay the log to inject the client requests into the application. A leader election component includes a heartbeat and the identity of the current leader. A permission management component allows a leader to request write permission to the local log while revoking the permission from other nodes.

RDMA connection endpoints are called Queue Pairs (QPs). Each QP is associated to a Completion Queue (CQ). Operations are posted to QPs as Work Requests (WRs). The RDMA hardware consumes the WR, performs the operation, and posts a Work Completion (WC) to the CQ. Applications make local memory available for remote access by registering local virtual memory regions (MRs) with the RDMA driver. Both QPs and MRs can have different access modes (e.g., read-only or read-write). The access mode is specified when initializing the QP or registering the MR, but can be changed later. MRs can overlap: the same memory can be registered multiple times, yielding multiple MRs, each with its own access mode. In this way, different remote machines can have different access levels to the same memory. The same effect can be obtained by using different access flags for the QPs used to communicate with remote machines.

3.3 Overview of Mu

3.3.1 Architecture

Figure 3.1 depicts the architecture of Mu. At the top, a client sends requests to an application and receives a response. We are not particularly concerned about how the client communicates with the application: it can use a network, a local pipe, a function call, etc. We do assume however that this communication is amenable to being captured and injected. That is, there is a mechanism to capture requests from the client before it reaches the application, so we can forward these requests to the replicas; a request is an opaque buffer that is not interpreted by Mu. Similarly, there is a mechanism to inject requests into the app. Providing

such mechanisms requires changing the application; however, in our experience, the changes are small and non-intrusive. These mechanisms are standard in any SMR system.

Each replica has an idea of which replica is currently the leader. A replica that considers itself the leader assumes that role (left of figure), and otherwise, assumes the role of a follower (right of figure). Each replica grants RDMA *write permission* to its log for its current leader and no other replica. The replicas constantly monitor their current leader to check that it is still active. The replicas might not agree on who the current leader is. But in *the common case*, all replicas have the same leader, and that leader is active. When that happens, Mu is simple and efficient. The leader captures a client request, uses an RDMA Write to append that request to the log of each follower, and then continues the application to process the request. When the followers detect a new request in their log, they inject the request into the application, thereby updating the replicas.

The main challenge in the design of SMR protocols is to handle leader failures. Of particular concern is the case when a leader appears failed (due to intermittent network delays) so another leader takes over, but the original leader is still active.

To detect failures in Mu, the leader periodically increments a local counter: the followers periodically check the counter using an RDMA Read. If the followers do not detect an increment of the counter after a few tries, a new leader is elected.

The new leader revokes a write permission by any old leaders, thereby ensuring that old leaders cannot interfere with the operation of the new leader. The new leader also reconstructs any partial work left by prior leaders.

Both the leader and the followers are internally divided into two major parts: the replication plane and the background plane. Roughly, the replication plane is responsible for copying requests captured by the leader to the followers, and replaying those requests at the followers' copy of the application. The background plane monitors (the health of) the leader and handles permission changes. Each plane has its own threads and queue pairs. This is in order to improve parallelism and provide isolation of performance and functionality. More specifically, the following components exist in each of the planes.

The replication plane has three components:

- *Replicator.* This component implements the main protocol to replicate a request from the leader to the followers, by writing the request in the followers' logs using RDMA Write.
- *Replayer.* This component replays entries from the local log.
- *Logging.* This component stores client requests to be replicated. Each replica has its own local log, which may be written remotely by other replicas according to previously granted permissions. Replicas also keep a copy of remote logs, which is used by a new

leader to reconstruct partial log updates by older leaders.

The background plane has two components:

- *Leader election.* This component detects failures of leaders and selects other replicas to become leader.
- *Permission management.* This component grants and revokes write access of local data by remote replicas. It maintains a permissions array, which stores access requests by remote replicas. Basically, a remote replica uses RDMA to store a 1 in this vector to request access.

We describe these planes in more detain in §3.4 and §3.5.

3.3.2 RDMA Communication

Each replica has two QPs for each remote replica: one QP for the replication plane and one for the background plane. The QPs for the replication plane share a completion queue, while the QPs for the background plane share another completion queue. The QPs operate in Reliable Connection (RC) mode.

Each replica also maintains two MRs, one for each plane. The MR of the replication plane contains the consensus log and the MR of the background plane contains metadata for leader election (§3.5.1) and permission management (§3.5.2). During execution, replicas may change the level of access to their log that they give to each remote replica; this is done by changing QP access flags. Note that all replicas always have remote read and write access permissions to the memory region in the background plane of each replica.

3.4 Replication Plane

The replication plane takes care of execution in the common case, but remains safe during leader changes. This is where we take care to optimize the latency of the common path. We do so by ensuring that, in the replication plane, only a leader replica communicates over the network, whereas all follower replicas are *silent* (i.e., only do local work).

In this section, we discuss algorithmic details related to replication in Mu. For pedagogical reasons, we first describe a basic version of the algorithm and then discuss extensions and optimizations to improve functionality and performance. We give the intuition why our algorithm is correct in this section and we provide a full correctness argument in Appendix B.

3.4.1 Basic Algorithm

The leader captures client requests, and calls *propose* to replicate these requests. It is simplest to understand our replication algorithm relative to the Paxos algorithm, which we briefly summarize; for details, we refer the reader to [146]. In Paxos, for each slot of the log, a leader first executes a *prepare phase* where it sends a proposal number to all replicas.[2] A replica replies with either *nack* if it has seen a higher proposal number, or otherwise with the value with the highest proposal number that it has accepted. After getting a majority of replies, the leader adopts the value with the highest proposal number. If it got no values (only acks), it adopts its own proposal value. In the next phase, the *accept phase*, the leader sends its proposal number and adopted value to all replicas. A replica acks if it has not received any prepare phase message with a higher proposal number.

In Paxos, replicas actively reply to messages from the leader, but in our algorithm, replicas are silent and communicate information passively by publishing it to their memory. Specifically, along with their log, a replica publishes a *minProposal* representing the minimum proposal number which it can accept. The correctness of our algorithm hinges on the leader reading and updating the minProposal number of each follower before updating anything in its log, and on updates on a replica's log happening in slot-order.

However, this by itself is not enough; Paxos relies on active participation from the followers not only for the data itself, but also to avoid races. Simply publishing the relevant data on each replica is not enough, since two competing leaders could miss each other's updates. This can be avoided if each of the leaders rereads the value after writing it [92]. However, this requires more communication. To avoid this, we shift the focus from the communication itself to the *prevention* of bad communication. A leader ℓ maintains a set of *confirmed followers*, which have granted write permission to ℓ and revoked write permission from other leaders before ℓ begins its operation. This is what prevents races among leaders in Mu. We describe these mechanisms in more detail below.

Log Structure. The main data structure used by the algorithm is the consensus log kept at each replica (Listing 3.1). The log consists of (1) a *minProposal* number, representing the smallest proposal number with which a leader may enter the accept phase on this replica; (2) a *first undecided offset (FUO)*, representing the lowest log index which this replica believes to be undecided; and (3) a sequence of slots—each slot is a (*propNr*, *value*) tuple.

Listing 3.1 – Log Structure.

```
struct Log {
  minProposal = 0,
  FUO = 0,
  slots[] = (0,⊥) for all slots }
```

[2] Paxos uses proposer and acceptor terms; instead, we use leader and replica.

Algorithm 3.2 – Basic Replication Algorithm of Mu.

```
1  Propose(myValue):
2      done = false
3      If I just became leader or I just aborted:
4          For every process p in parallel:
5              Request permission from p
6              If p acks, add p to confirmedFollowers
7          Until this has been done for a majority
8      While not done:
9          Execute Prepare Phase
10         Execute Accept Phase

12 Prepare Phase:
13     For every process p in confirmedFollowers:
14         Read minProposal from p's log
15     Pick a new proposal number, propNum, higher than any minProposal seen
        ↪ so far
16     For every process p in confirmedFollowers:
17         Write propNum into LOG[p].minProposal
18         Read LOG[p].slots[myFUO]
19         Abort if any write fails
20     if all entries read were empty:
21         value = myValue
22     else:
23         value = entry value with the largest proposal number of slots
            ↪ read

25 Accept Phase:
26     For every process p in confirmedFollowers:
27         Write value,propNum to p in slot myFUO
28         Abort if any write fails
29     If value == myValue:
30         done = true
31     Locally increment myFUO
```

Algorithm Description. Each leader begins its propose call by constructing its *confirmed followers* set (Algorithm 3.2, lines 4–7). This step is only necessary the first time a new leader invokes propose or immediately after an abort. This step is done by sending permission requests to all replicas and waiting for a majority of acks. When a replica acks, it means that this replica has granted write permission to this leader and revoked it from other replicas. The leader then adds this replica to its confirmed followers set. During execution, if the leader ℓ fails to write to one of its confirmed followers, because that follower crashed or gave write access to another leader, ℓ aborts and, if it still thinks it is the leader, it calls propose again.

After establishing its confirmed followers set, the leader invokes the prepare phase. To do so, the leader reads the *minProposal* from its confirmed followers (line 14) and chooses a proposal number propNum which is larger than any that it has read or used before. Then, the leader writes its proposal number into *minProposal* for each of its confirmed followers. Recall that if this write fails at any follower, the leader aborts. It is safe to overwrite a follower f's

minProposal in line 17 because, if that write succeeds, then ℓ has not lost its write permission since adding f to its confirmed followers set, meaning no other leader wrote to f since then. To complete its prepare phase, the leader reads the relevant log slot of all of its confirmed followers and, as in Paxos, adopts either (a) the value with the highest proposal number, if it read any non-$\perp$ values, or (b) its own initial value, otherwise.

The leader ℓ then enters the accept phase, in which it tries to commit its previously adopted value. To do so, ℓ writes its adopted value to its confirmed followers. If these writes succeed, then ℓ has succeeded in replicating its value. No new value or minProposal number could have been written on any of the confirmed followers in this case, because that would have involved a loss of write permission for ℓ. Since the confirmed followers set constitutes a majority of the replicas, this means that ℓ's replicated value now appears in the same slot at a majority.

Finally, ℓ increments its own FUO to denote successfully replicating a value in this new slot. If the replicated value was ℓ's own proposed value, then it returns from the *propose* call; otherwise it continues with the prepare phase for the new FUO.

3.4.2 Extensions

The basic algorithm described so far is clear and concise, but it also has downsides related to functionality and performance. We now address these downsides with some extensions, all of which are standard for Paxos-like algorithms; their correctness is discussed in the supplementary material.

Bringing stragglers up to date. In the basic algorithm, if a replica r is not included in some leader's confirmed followers set, then its log will lag behind. If r later becomes leader, it can catch up by proposing new values at its current FUO, discovering previously accepted values, and re-committing them. This is correct but inefficient. Even worse, if r never becomes leader, then it will never recover the missing values. We address this problem by introducing an update phase for new leaders. After a replica becomes leader and establishes its confirmed followers set, but before attempting to replicate new values, the new leader (1) brings itself up to date with its highest-FUO confirmed follower and (2) brings its followers up to date. This is done by copying the contents of the more up-to-date log to the less up-to-date log.

Followers commit in background. In the basic algorithm, followers do not know when a value is committed and thus cannot replay the requests in the application. This is easily fixed without additional communication. Since a leader will not start replicating in an index i before it knows index $i-1$ to be committed, followers can monitor their local logs and commit all values up to (but excluding) the highest non-empty log index. This is called *commit piggybacking*, since the commit message is folded into the next replicated value.

Omitting the prepare phase. Once a leader finds only empty slots at a given index at all of its confirmed followers at line 18, then no higher index may contain an accepted value at any confirmed follower; thus, the leader may omit the prepare phase for higher indexes (until it aborts, after which the prepare phase becomes necessary again). This optimization concerns performance on the common path. With this optimization, the cost of a Propose call becomes a single RDMA write to a majority in the common case.

Growing confirmed followers. In our algorithm, the confirmed followers set remains fixed after the leader initially constructs it. This implies that processes outside the leader's confirmed followers set will miss updates, even if they are alive and timely, and that the leader will abort even if one of its followers crashes. To avoid this problem, we extend the algorithm to allow the leader to grow its confirmed followers set by adding replicas which respond to its initial request for permission. The leader must bring these replicas up to date before adding them to its set. When its confirmed follower set is large, the leader cannot wait for its RDMA reads and writes to complete at all of its confirmed followers before continuing, since we require the algorithm to continue operating despite the failure of a minority of the replicas; instead, the leader waits for just a majority of the replicas to complete.

Replayer. Followers continually monitor the log for new entries. This creates a challenge: how to ensure that the follower does not read an incomplete entry that has not yet been fully written by the leader. We adopt a standard approach: we add an extra *canary byte* at the end of each log entry [162, 225]. Before issuing an RDMA Write to replicate a log entry, the leader sets the entry's canary byte to a non-zero value. The follower first checks the canary and then the entry contents. In theory, it is possible that the canary gets written before the other contents under RDMA semantics. In practice, however, NICs provide left-to-right semantics in certain cases (e.g., the memory region is in the same NUMA domain as the NIC), which ensures that the canary is written last. This assumption is made by other RDMA systems [84, 85, 132, 162, 225]. Alternatively, we could store a checksum of the data in the canary, and the follower could read the canary and wait for the checksum to match the data.

3.5 Background Plane

The background plane has two main roles: electing and monitoring the leader, and handling permission change requests. In this section, we describe these mechanisms.

3.5.1 Leader Election

The *leader election component* of the background plane maintains an estimate of the current leader, which it continually updates. The replication plane uses this estimate to determine whether to execute as leader or follower.

Each replica independently and locally decides who it considers to be leader. We opt for a simple rule: replica i decides that j is leader if j is the replica with the lowest id, among those that i considers to be alive.

To know whether a replica has failed, we employ a *pull-score* mechanism, based on a *local heartbeat* counter. A leader election thread continually increments its own counter locally and uses RDMA Reads to read the counters (heartbeats) of other replicas and check whether they have been updated. It maintains a *score* for every other replica. If a replica has updated its counter since the last time it was read, we increment that replica's score; otherwise, we decrement it. Once a replica's score drops below a threshold, we consider it to have failed. To avoid oscillation, we have different *failure* and *recovery* thresholds, chosen so as to avoid false positives.

3.5.2 Permission Management

The permission management module is used when changing leaders. Each replica maintains the invariant that only one replica at a time has write permission on its log. As explained in Section 3.4, when a leader changes in Mu, the new leader must request write permission from all the other replicas; this is done through a simple RDMA Write to a *permission request array* on the remote side. When a replica r sees a *permission request* from a would-be leader ℓ, r revokes write access from the current holder, grants write access to ℓ, and sends an ack to ℓ.

During the transition phase between leaders, it is possible that several replicas think themselves to be leader, and thus the permission request array may contain multiple entries. A permission management thread monitors and handles permission change requests one by one in order of requester id by spinning on the local permission request array.

RDMA provides multiple mechanisms to grant and revoke write access. The first mechanism is to register the consensus log as multiple overlapping RDMA memory regions (MRs), one per remote replica. In order to grant or revoke access from replica r, it suffices to re-register the MR corresponding to r with different access flags. The second mechanism is to revoke r's write access by moving r's QP to a non-operational state (e.g., *init*); granting r write access is then done by moving r's QP back to the *ready-to-receive (RTR)* state. The third mechanism is to grant or revoke access from replica r by changing the access flags on r's QP.

We compare the performance of these three mechanisms in Figure 3.2, as a function of the log size (which is the same as the RDMA MR size). We observe that the time to re-register an RDMA MR grows with the size of the MR, and can reach values close to 100ms for a log size of 4GB. On the other hand; the time to change a QPs access flags or cycle it through different states is independent of the MR size, with the former being roughly 10 times faster than the latter. However, changing a QPs access flags while RDMA operations to that QP are in flight sometimes causes the QP to go into an error state. Therefore, in Mu we use a fast-slow path approach: we first optimistically try to change permissions using the faster QP access flag

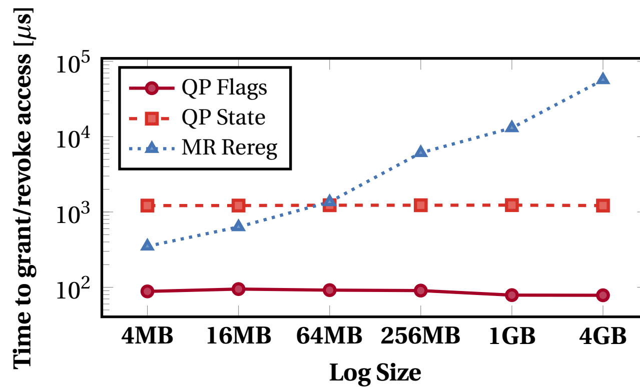


Figure 3.2 – Performance comparison of different permission switching mechanisms. *QP Flags*: change the access flags on a QP; *QP Restart*: cycle a QP through the *reset*, *init*, *RTR* and *RTS* states; *MR Rereg*: re-register an RDMA MR with different access flags.

method and, if that leads to an error, switch to the slower, but robust, QP state method.

3.5.3 Log Recycling

Conceptually, a log is an infinite data structure but in practice we need to implement a circular log with limited memory. This is done as follows. Each follower has a local *log head* variable, pointing to the first entry not yet executed in its copy of the application. The replayer thread advances the log head each time it executes an entry in the application. Periodically, the leader's background plane reads the log heads of all followers and computes *minHead*, the minimum of all log head pointers read from the followers. Log entries up to the minHead can be reused. Before these entries can be reused, they must be zeroed out to ensure the correct function of the canary byte mechanism. Thus, the leader zeroes all follower logs after the leader's first undecided offset and before minHead, using an RDMA Write per follower. Note that this means that a new leader must first execute all leader change actions, ensuring that its first undecided offset is higher than all followers' first undecided offsets, before it can recycle entries. To facilitate the implementation, we ensure that the log is never completely full.

3.5.4 Adding and Removing Replicas

Mu adopts a standard method to add or remote replicas: use consensus itself to inform replicas about the change [146]. More precisely, there is a special log entry that indicates that replicas have been removed or added. Removing replicas is easy: once a replica sees it has been removed, it stops executing, while other replicas subsequently ignore any communication with it. Adding replicas is more complicated because it requires copying the state of an existing replica into the new one. To do that, Mu uses the standard approach of check-pointing state, and we do that from one of the followers [225].

3.6 Implementation

Mu is implemented in 7157 lines of C++ code (CLOC [71]). It uses the *ibverbs* library for RDMA over Infiniband. We implemented all features and extensions in sections 3.4 and 3.5, except adding/removing replicas. Moreover, we implement some standard RDMA optimizations to reduce latency. RDMA Writes and Sends with payloads below a device-specific limit (256 bytes in our setup) are inlined, meaning that their payload is written directly to their work request. We pin threads to cores in the NUMA node of the NIC.

3.7 Evaluation

Our goal is to evaluate whether Mu indeed provides viable replication for microsecond computing. We aim to answer the following questions in our evaluation:

- What is the replication latency of Mu? How does it change with payload size and the application being replicated? How does Mu compare to other solutions?
- What is Mu's fail-over time?
- What is the throughput of Mu?

We evaluate Mu on a 4-node cluster, where each node has two Intel Xeon E5-2640 v4 CPUs @ 2.40GHz (20 cores, 40 threads total per node), 256 GB of RAM equally split across the two NUMA domains, and a Mellanox Connect-X 4 NIC. All 4 nodes are connected to a single 100 Gbps Mellanox MSB7700 switch through 100 Gbps Infiniband. All experiments show 3-way replication, which accounts for most real deployments [119]. With more replicas (results omitted for brevity), replication latencies increases gradually with the number of replicas, up to 35% higher for Mu (for 9 replicas) and a larger increase than Mu for other systems at every replication level.

We compare against APUS [225], DARE [199], and Hermes [136] where possible. The most recent system, HovercRaft [140], also provides SMR but its latency at 30–60 microseconds is substantially higher than the other systems, so we do not consider it further. For a fair comparison, we disable APUS's persistence to stable storage, since Mu, DARE, and Hermes all provide only in-memory replication.

We measure time using the POSIX `clock_gettime` function, with the `CLOCK_MONOTONIC` parameter. In our deployment, the resolution and overhead of `clock_gettime` is around $16ns$ [82]. In our figures, we show bars labeled with the median latency, with error bars showing 99-percentile and 1-percentile latencies. These statistics are computed over 1 million samples with a payload of 64-bytes each, unless otherwise stated.

Applications. We use Mu to replicate several microsecond apps: three key-value stores, as well as an order matching engine for a financial exchange.

The key-value stores that we replicate with Mu are Redis [206], Memcached [176], and HERD [132]. For the first two, the client is assumed to be on a different cluster, and connects to the servers over TCP. In contrast, HERD is a microsecond-scale RDMA-based key-value store. We replicate it over RDMA and use it as an example of a microsecond application. Integration with the three applications requires 183, 228 and 196 additional lines of code, respectively.

The other app is in the context of financial exchanges, in which parties unknown to each other submit buy and sell orders of stocks, commodities, derivatives, etc. At the heart of a financial exchange is an order matching engine [14], such as Liquibook [161], which is responsible for matching the buy and sell orders of the parties. We use Mu to replicate Liquibook. Liquibook's input are buy and sell orders. We created an unreplicated client-server version of Liquibook using eRPC [131], and then replicated this system using Mu. The eRPC integration and the replication required 611 lines of code in total.

3.7.1 Common-Case Latency

We begin by testing the overhead that Mu introduces in normal execution, when there is no leader failure. For these experiments, we first measure raw replication latency and compare Mu to other replication systems, as well as to itself under different payloads and attached applications. We then evaluate client-to-client application latency.

Effect of Payload and Application on Latency. We first study Mu in isolation, to understand its replication latency under different conditions.

We evaluate the raw replication latency of Mu in two settings: *standalone* and *attached*. In the standalone setting, Mu runs just the replication layer with no application and no client; the leader simply generates a random payload and invokes `propose()` in a tight loop. In the attached setting, Mu is integrated into one of a number of applications; the application client produces a payload and invokes `propose()` on the leader. These settings could be different as Mu and the application could interfere with each other.

Figure 3.3 compares standalone to attached runs as we vary payload size. Liquibook and Herd allow only one payload size (32 and 50 bytes), so they have only one bar each in the graph, while Redis and Memcached have many bars.

We see that the standalone version slightly outperforms the attached runs, for all tested applications and payload sizes. This is due to processor cache effects; in standalone runs, replication state, such as log and queue pairs, are always in cache, and the requests themselves need not be fetched from memory. This is not the case when attaching to an application. Mu supports two ways of attaching to an application, which have different processor cache

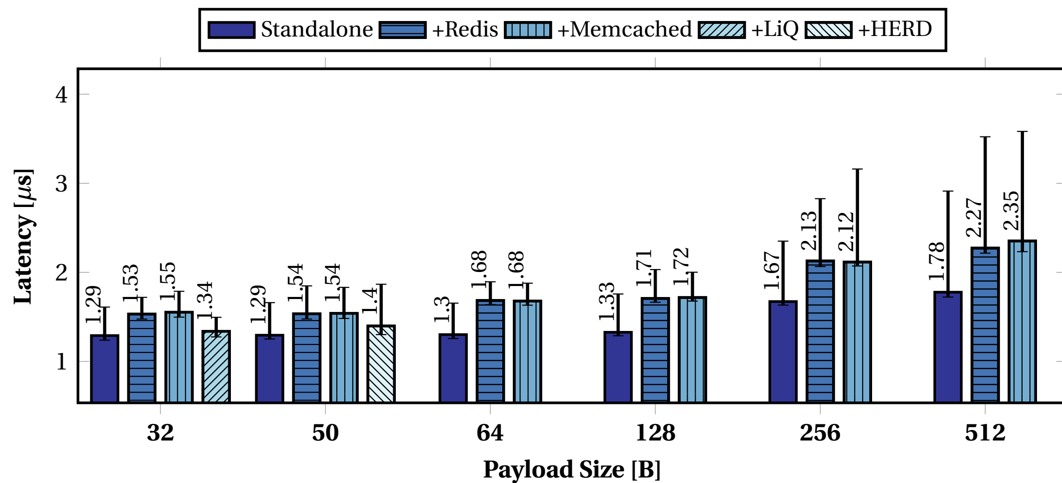


Figure 3.3 – Replication latency of Mu integrated into different applications [Memcached (mcd), Liquibook (LiQ), Redis (rds), HERD] and payload sizes.

sharing. The *direct* mode uses the same thread to run both the application and the replication, and so they share L1 and L2 caches. In contrast, the *handover* method places the application thread on a separate core from the replication thread, thus avoiding sharing L1 or L2 caches. Because the application must communicate the request to the replication thread, the handover method requires a cache coherence miss per replicated request. This method consistently adds ≈400ns over the standalone method. For applications with large requests, this overhead might be preferable to the one caused by the direct method, where replication and application compete for CPU time. For lighter weight applications, the direct method is preferable. In our experiments, we measure both methods and show the best method for each application: Liquibook and HERD use the direct method, while Redis and Memcached use the handover method.

We see that for payloads under 256 bytes, standalone latency remains constant despite increasing payload size. This is because we can RDMA-inline requests for these payload sizes, so the amount of work needed to send a request remains the same. At a payload of 256 bytes, the NIC must do a DMA itself to fetch the value to be sent, which incurs a gradual increase in overhead as the payload size increases. However, we see that Mu still performs well even at larger payloads quite well; at 512B, the median latency is only 35% higher than the latency of inlined payloads.

Comparing Mu to Other Replication Systems. We now study the replication time of Mu compared to other replication systems, for various applications. This comparison is not possible for every pair of replication system and application, because certain replication systems are incompatible with certain applications. In particular, APUS works only with socket-based applications (Memcached and Redis). In DARE and Hermes, the replication protocol is bolted onto a key-value store, so we cannot attach it to the apps we consider—

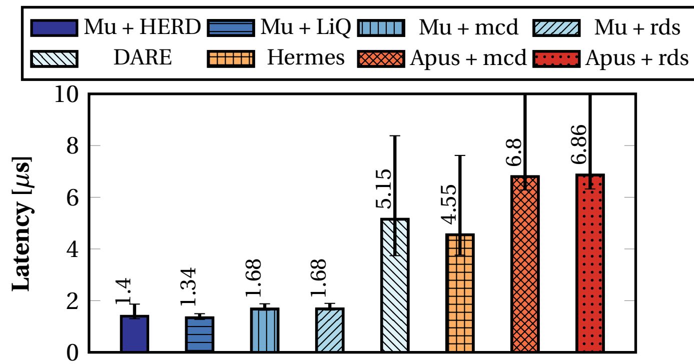


Figure 3.4 – Replication latency of Mu compared with other replication solutions: DARE, Hermes, Apus on memcached (mcd), and Apus on Redis (rds).

instead, we report their performance with their key-value stores.

Figure 3.4 shows the replication latencies of these systems. Mu's median latency outperforms all competitors by at least 2.7×, outperforming APUS on the same applications by 4×. Furthermore, Mu has smaller tail variation, with a difference of at most 500ns between the 1-percentile and 99-percentile latency. In contrast, Hermes and DARE both varied by more than $4\mu s$ across our experiments, with APUS exhibiting 99-percentile executions up to $20\mu s$ slower (cut off in the figure). We attribute this higher variance to two factors: the need to involve the CPU of many replicas in the critical path (Hermes and APUS), and sequentializing several RDMA operations so that their variance aggregates (DARE and APUS).

End-to-End Latencies. Figure 3.5 shows the end-to-end latency of our tested applications, which includes the latency incurred by the application and by replication (if enabled). We show the result in three graphs corresponding to three classes of applications.

The leftmost graph is for Liquibook. The left bar is the unreplicated version, and the right bar is replicated with Mu. We can see that the median latency of Liquibook without replication is $4.08\mu s$, and therefore the overhead of replication is around 35%. There is a large variance in latency, even in the unreplicated system. This variance comes from the client-server communication of Liquibook, which is based on eRPC. This variance changes little with replication. The other replication systems cannot replicate Liquibook (as noted before, DARE and Hermes are bolted onto their app, and APUS can replicate only socket-based applications). However, extrapolating their latency from Figure 3.4, they would add unacceptable overheads—over 100% overhead for the best alternative (Hermes).

The middle graph in Figure 3.5 shows the client-to-client latency of replicated and unreplicated microsecond-scale key-value stores. The first bars in orange shows HERD unreplicated and HERD replicated by Mu. The green bar shows DARE's key-value store with its own replication system. The median unreplicated latency of HERD is $2.25\mu s$, and Mu adds $1.34\mu s$. While this is a significant overhead (59% of the original latency), this overhead is lower than any alternative.

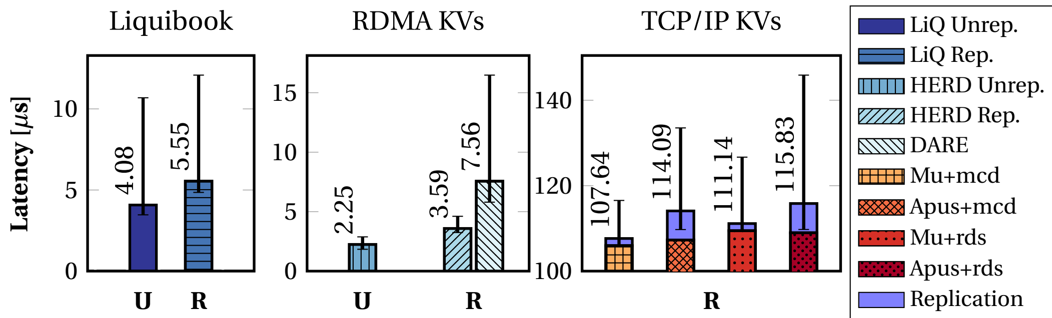


Figure 3.5 – End-to-end latencies of applications, unreplicated (U) and replicated (R). The first graph shows a financial exchange app (Liquibook) unreplicated and replicated with Mu. The second graph shows microsecond key-value stores: HERD unreplicated, HERD replicated with Mu, and DARE. The third graph shows traditional key-value stores: Memcached and Redis, replicated with Mu and APUS. In this graph, each bar is split in two parts: application latency (bottom) and replication latency (top).

We do not show Hermes in this graph since Hermes does not allow for a separate client, and only generates its requests on the servers themselves. HERD replicated with Mu is the best option for a replicated key-value store, with overall median latency 2× lower than the next best option, with a much lower variance.

The rightmost graph in Figure 3.5 shows the replication of the traditional key-value stores, Redis and Memcached. For these applications, we compare replication with Mu to replication with APUS. Each bar has two parts: the bottom is the latency of the application and client-server communication, and the top is the replication latency. Note that the scale starts at $100\mu s$ to show better precision.

Mu replicates these apps about $5\mu s$ faster than APUS, a 5% difference. With a faster network, this difference would be bigger. In either case, Mu provides fault tolerant replication with essentially no overhead for these applications.

3.7.2 Fail-Over Time

We now study Mu's fail-over time. In these experiments, we run the system and subsequently introduce a leader failure. To get a thorough understanding of the fail-over time, we repeatedly introduce leader failures (1000 times) and plot a histogram of the fail-over times we observe. We also time the latency of permission switching, which corresponds to the time to change leaders after a failure is detected. The detection time is the difference between the total fail-over time and the permission switch time.

We inject failures by delaying the leader, thus making it become temporarily unresponsive. This causes other replicas to observe that the leader's heartbeat has stopped changing, and thus detect a failure.

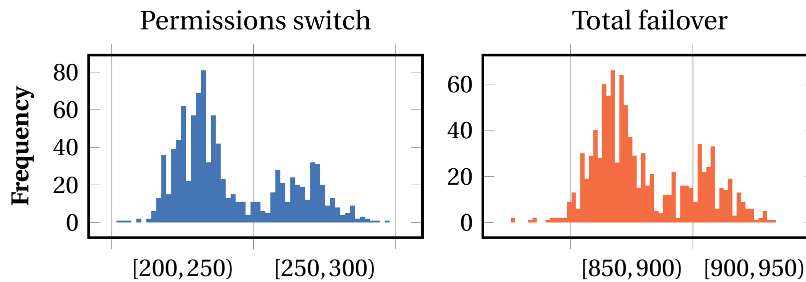


Figure 3.6 – Fail-over time distribution in Mu.

Figure 3.6 shows the results. We first note that the total fail-over time is quite low; the median fail-over time is 873μs and the 99-percentile fail-over time is 947μs, still below a millisecond. This represents an order of magnitude improvement over the best competitor at ≈10 ms (HovercRaft [140]).

The time to switch permissions constitutes about 30% of the total fail-over time, with mean latency at 244μs, and 99-percentile at 294μs. Recall that this measurement in fact encompasses two changes of permission at each replica; one to revoke write permission from the old leader and one to grant it to the new leader. Thus, improvements in the RDMA permission change protocol would be doubly amplified in Mu's fail-over time.

The rest of the fail-over time is attributed to failure detection (≈600μs). Although our pull-score mechanism does not rely on network variance, there is still variance introduced by process scheduling (e.g., in rare cases, the leader process is descheduled by the OS for tens of microseconds)—this is what prevented us from using smaller timeouts/scores and it is an area under active investigation for microsecond apps [36, 190, 202, 204].

3.7.3 Throughput

While Mu optimizes for low latency, in this section we evaluate the throughput of Mu. In our experiment, we run a standalone microbenchmark (not attached to an application). We increase throughput in two ways: by batching requests together before replicating, and by allowing multiple outstanding requests at a time. In each experiment, we vary the maximum number of outstanding requests allowed at a time, and the batch sizes.

Figure 3.7 shows the results in a latency-throughput graph. Each line represents a different max number of outstanding requests, and each data point represents a different batch size. As before, we use 64-byte requests.

We see that Mu reaches high throughput with this simple technique. At its highest point, the throughput reaches 47 Ops/μs with a batch size of 128 and 8 concurrent outstanding requests, with per-operation median latency at 17μs. Since the leader is sending requests to two other replicas, this translates to a throughput of 48Gbps, around half of the bandwidth of the NIC.

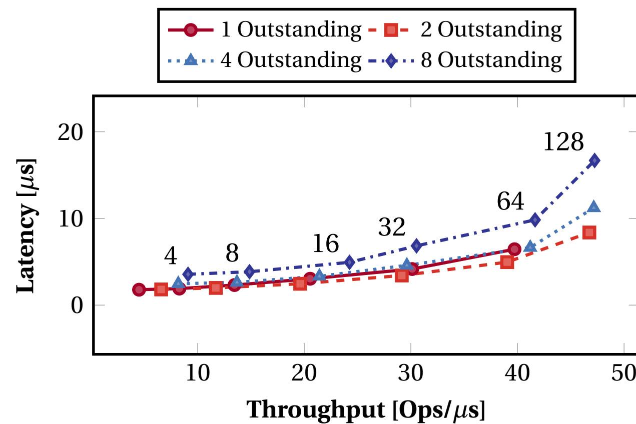


Figure 3.7 – Latency vs throughput in Mu. Each line represents a different number of allowed concurrent outstanding requests. Each point on the lines represents a different batch size. Batch size shown as annotation close to each point.

Latency and throughput both increase as the batch size increases. Median latency is also higher with more concurrent outstanding requests. However, the latency increases slowly, remaining at under $10\mu s$ even with a batch size of 64 and 8 outstanding requests.

There is a throughput wall at around 45 Ops/μs, with latency rising sharply. This can be traced to the transition between the client requests and the replication protocol at the leader replica. The leader must copy the request it receives into a memory region prepared for its RDMA write. This memory operation becomes a bottleneck. We could optimize throughput further by allowing direct contact between the client and the follower replicas. However, that may not be useful as the application itself might need some of the network bandwidth for its own operation, so the replication protocol should not saturate the network.

Increasing the number of outstanding requests while keeping the batch size constant substantially increases throughput at a small latency cost. The advantage of more outstanding requests is largest with two concurrent requests over one. Regardless of batch size, this allows substantially higher throughput at a negligible latency increase: allowing two outstanding requests instead of one increases latency by at most $400ns$ for up to a batch size of 32, and only $1.1\mu s$ at a batch size of 128, while increasing throughput by 20–50% depending on batch size. This effect grows less pronounced with higher numbers of outstanding requests.

Similarly, increasing batch size increases throughput with a low latency hit for small batch sizes, but the latency hit grows for larger batches. Notably, using 2 outstanding requests and a batch size of 32 keeps the median latency at only $3.4\mu s$, but achieves throughput of nearly 30 Ops/μs.

3.8 Related Work

SMR in General. State machine replication is a common technique for building fault-tolerant, highly available services [146, 211]. Many practical SMR protocols have been designed, addressing simplicity [34, 45, 119, 151, 187], cost [142, 149], and harsher failure assumptions [49, 50, 92, 142]. In the original scheme, which we follow, the order of all operations is agreed upon using consensus instances. At a high-level, our Mu protocol resembles the classical Paxos algorithm [146], but there are some important differences. In particular, we leverage RDMA's ability to grant and revoke access permissions to ensure that two leader replicas cannot both write a value without recognizing each other's presence. This allows us to optimize out participation from the follower replicas, leading to better performance. Furthermore, these dynamic permissions guide our unique leader changing mechanism.

Several implementations of multipaxos avoid repeating Paxos's prepare phase for every consensus instance, as long as the same leader remains [53, 147, 169]. Piggybacking a commit message onto the next replicated request, as is done in Mu, is also used as a latency-hiding mechanism in [169, 225].

Aguilera et al. [5] suggested the use of local heartbeats in a leader election algorithm designed for a theoretical message-and-memory model, in an approach similar to our pull-score mechanism. However, no system has so far implemented such local heartbeats for leader election in RDMA.

Single round-trip replication has been achieved in several previous works using two-sided sends and receives [86, 136, 137, 142, 149]. Our theoretical work in Chapter 2 has shown that single-shot consensus can be achieved in a single one-sided round trip. However, Mu is the first system to put that idea to work and implement *one-sided single round-trip* SMR.

Alternative reliable replication schemes totally order only non-conflicting operations [62, 117, 136, 148, 192, 195, 213]. These schemes require opening the service being replicated to identify which operations commute. In contrast, we designed Mu assuming the replicated service is a black box. If desired, several parallel instances of Mu could be used to replicate concurrent operations that commute. This could be used to increase throughput in specific applications.

It is also important to notice that we consider "crash" failures. In particular, we assume nodes cannot behave in a Byzantine manner [49, 61, 142].

Improving the Stack Underlying SMR. While we propose a new SMR algorithm adapted to RDMA in order to optimize latency, other systems keep a classical algorithm but improve the underlying communication stack [131, 159]. With this approach, somehow orthogonal to ours, the best reported replication latency is 5.5 μs [131], almost 5× slower than Mu. HovercRaft [140] shifts the SMR from the application layer to the transport layer to avoid IO and CPU bottlenecks on the leader replica. However, their request latency is more than an order of

magnitude more than that of Mu, and they do not optimize fail-over time.

Some SMR systems leverage recent technologies such as programmable switches and NICs [125, 130, 160, 164]. However, programmable networks are not as widely available as RDMA, which has been commoditized with technologies such as RoCE and iWARP.

SMR over RDMA. A few SMR systems have recently been designed for RDMA [129, 199, 225]. DARE [199] was the first RDMA-based SMR system. Similarly to Mu, DARE uses only one-sided RDMA verbs executed by the leader to replicate the log in normal execution. However, DARE requires updating the tail pointer of each replica's log in a separate RDMA Write from the one that copies over the new value, and therefore induces more round-trips for replication. Furthermore, DARE has a heavier leader election protocol than Mu's. DARE's leader election is similar to the one used in RAFT, in which care is taken to ensure that at most one process considers itself leader at any point in time. APUS [225] improved upon DARE's throughput. However, APUS requires active participation from the follower replicas during the replication protocol, resulting in higher latencies. Both DARE and APUS use transitions through queue pair states to allow or deny RDMA access. This is reminiscent of our permissions approach, but is less fine grained.

Derecho [129] provides durable and non-durable SMR, by combining a data movement protocol (SMC or RDMC) with a shared-state table primitive (SST) for determining when it is safe to deliver messages. This design yields high throughput but also high latency: a minimum of $10\mu s$ for non-durable SMR [129, Figure 12(b)] and more for durable SMR. This latency results from a node delaying the delivery of a message until all nodes have confirmed its receipt using the SST, which takes additional RDMA communication steps compared to Mu. It would be interesting to explore how Mu's protocol could improve a large system like Derecho.

Other RDMA Applications. More generally, RDMA has recently been the focus of many data center system designs, including key-value stores [84, 132] and transactions [134, 227]. Kalia et al. provide guidelines on the best ways to use RDMA to enhance performance [133]. Many of their suggested optimizations are employed by Mu. Kalia et al. also advocate the use of two-sided RDMA verbs (Sends/Receives) instead of RDMA Reads in situations in which a single RDMA Read might not suffice. However, this does not apply to Mu, since we know a priori which memory location to read, and we rarely have to follow up with another read.

Failure detection. Failure detection is typically done using timeouts. Conventional wisdom is that timeouts must be large, in the seconds [156], though some systems report timeouts as low as 10 milliseconds [140]. It is possible to improve detection time using inside information [154, 156] or fine-grained reporting [155], which requires changes to apps and/or the infrastructure. This is orthogonal to our score-based mechanism and could be used to further improve Mu.

3.9 Conclusion

Computers have progressed from batch-processing systems that operate at the time scale of minutes, to progressively lower latencies in the seconds, then milliseconds, and now we are in the microsecond revolution. Work has already started in this space at various layers of the computing stack. Our contribution fits in this context, by providing generic microsecond replication for microsecond apps.

Mu is a state machine replication system that can replicate microsecond applications with little overhead. This involved two goals: achieving low latency on the common path, and minimizing fail-over time to maintain high availability. To reach these goals, Mu relies on (a) RDMA permissions to replicate a request with a single one-sided operation, as well as (b) a failure detection mechanism that does not incur false positives due to network delays—a property that permits Mu to use aggressively small timeout values.

Sharing Persistent Memory Part II

4 The Inherent Cost of Remembering Consistently

Persistent memory (PM) promises fast, byte-addressable and durable storage, with raw access latencies in the same order of magnitude as DRAM. But in order to take advantage of the durability of PM, programmers need to design *persistent* objects which maintain consistent state across system crashes and restarts. Concurrent implementations of persistent objects typically make heavy use of expensive persistent fence instructions to order PM accesses, thus negating some of the performance benefits of PM.

This raises the question of the minimal number of persistent fence instructions required to implement a persistent object. We answer this question in the deterministic lock-free case by providing lower and upper bounds on the required number of fence instructions. We obtain our upper bound by presenting a new universal construction that implements durably any object using at most one persistent fence per update operation invoked. Our lower bound states that in the worst case, each process needs to issue at least one persistent fence per update operation invoked.

4.1 Introduction

Persistent memory (PM) is fast, byte-addressable memory that preserves its contents even in the absence of power. Recent years have seen significant research into PM [152, 181, 205, 218], but the technology is only now starting to become commercially available.

PM shares similarities with both traditional stable storage and DRAM. Like stable storage, PM allows programs to persist their state across power failures or machine restarts. Unlike stable storage, PM is byte-addressable and fast (with access times in the same order of magnitude as DRAM [230]). In this sense, PM promises applications that are durable and fast, for they would not need to access slow storage to persist their state or during recovery.

Yet, the task of designing fast persistent objects (as building blocks of persistent applications) is complicated by two factors:

1. Processor registers and caches are expected to remain volatile (transient) for the foreseeable future. Therefore, simply writing to a memory location is not sufficient to ensure the persistence of its contents (even if the memory location is in PM), because the write instruction might, for instance, be satisfied in the cache and thus lost in the case of a crash.

2. There is a priori no guarantee on the order in which cache lines are written back to PM. However, program correctness might rely on such a guarantee, especially in a concurrent setting, which is the focus of this chapter.

Due to these two factors, programming for PM requires the use of flush instructions to force cache line write-backs, as well as of expensive fence instructions to ensure ordering among such flushes.

As we explain in Section 4.2, it is these latter fences that dominate the cost of PM write-backs, which raises an interesting question: What is the minimum number of persistent fences required to implement a persistent object? In this chapter, we answer this question for concurrent lock-free objects, by providing both upper and lower bounds on the number of fences required to implement them persistently.

We focus on the lock-free case because it provides an interesting trade-off. On the one hand, intuitively, lock-free objects can be implemented with a small number of fences, because they are already required to always be in a consistent state, such that progress can be made despite the failure of any number of processes. A priori, durability has the related requirement of an object state being consistent, no matter when a crash may occur. On the other hand, it is this very need for consistency that makes lock-free objects require at least a minimal number of fences for each operation invoked, as we show in the chapter (we discuss lock-based objects in Section 5.9).

The correctness (safety) property we consider in this chapter is durable linearizability [127]. Durably linearizable objects satisfy the standard linearizability property: every operation seems to happen instantaneously at a *linearization point* between its invocation and response, in separation from any other process in the system. In addition, after a full-system crash, the state of the object must reflect a consistent operation subhistory that includes all operations completed by the time of the crash.

For the upper bound, we propose a new universal construction called *Order Now, Linearize Later* (ONLL) that takes a deterministic sequential specification of an object *O* and produces a lock-free durably linearizable implementation of *O* that uses at most one fence per operation invoked, in the worst case. Our construction in fact guarantees detectable execution [91], an even stronger property than durable linearizability, which ensures in addition that, upon recovery, processes can determine which operations were linearized before the crash and which operations were not.

In our universal construction, we distinguish between read-only and update operations. An update operation op proceeds in 3 steps, called *order*, *persist* and *linearize*, respectively. First, op synchronizes with other update operations to establish the linearization order of op. This step uses a shared lock-free execution trace data structure, based on a lock-free queue, for determining this order in a lock-free manner. Second, op is stored in PM by using a per-process persistent log. Crucial to this construction is the fact that the persistent log can be implemented with only one persistent fence per append operation [64]. A helping mechanism is used to ensure that delayed processes do not create inconsistencies in the state of the object. Third, op announces that it has completed the persistence step. This is also the linearization point [116] of op if it runs solo. When setting the linearization point, care is taken to respect the linearization order computed in Step 1. A read-only operation determines its return value based on the update that most recently announced completion of the persistence step.

Since persistent fences are only performed when appending one or more updates to a process' persistent log, it is clear that our construction uses at most one persistent fence per operation. Moreover, no process can prevent the system from making progress, thus the construction is lock-free.

Our lower bound states that any lock-free implementation of a persistent object has at least one execution in which all concurrent processes need to issue one fence instruction per update operation invoked. The intuition behind this result is that processes cannot always rely on each other to persist updates and must therefore sometimes persist these updates themselves. To see this, imagine that some process p is designated to persist updates for one or more other processes but p is delayed. In order for lock-freedom to be satisfied, those other processes cannot wait indefinitely for p, and so must persist their updates themselves, thus each incurring the cost of persistent fences.

To summarize, the contributions of this chapter are:

1. The ONLL universal construction, providing a lock-free durably linearizable implementation of any deterministic object. ONLL uses a single persistent fence per update operation and no persistent fences for read-only operations. ONLL also serves as upper bound on the number of persistent fences required to implement such objects.
2. A lower bound on the number of persistent fences in a lock-free durably linearizable implementation of an object.

We also discuss extensions to our universal construction for wait-freedom, improved read performance and memory reclamation.

The rest of this chapter is organized as follows. Section 4.2 recalls useful background. In Section 4.3, we give a high-level overview of our universal construction. In Section 4.4, we describe in detail the universal construction algorithm and we prove its correctness in Section 4.5. In Section 4.6, we present our lower bound result. We discuss relevant related work

in Section 4.7 and conclude with an overview of possible extensions and future directions in Section 5.9.

4.2 Background

4.2.1 Persistent Memory

So far, storage has been either slow but durable (e.g., SSD, hard disk, magnetic tape) or fast but volatile (e.g., DRAM). PM promises to combine the best of both worlds through fast and durable storage. Several implementations of PM are foreseen: Memristors [218], Phase Change Memory [152, 205] and 3D XPoint [181].

PM is expected to be byte-addressable and attached directly to the memory bus of the CPU, accessible by standard load and store instructions. Thus, programming for PM will probably be closer to programming for DRAM than for block-based storage. As argued in the introduction, the main difficulty in programming for PM will likely stem from the fact that a priori there is no guarantee on when and in what order (volatile) cache lines will be written back to PM. Therefore, programmers will need to use special instructions to ensure cache lines are written to the PM.

One such instruction on Intel machines is `clflush` [122], which forces a cache-line to be written back to the PM. This instruction is strongly ordered: a call to `clflush` returns only once the cache line is written-back to the PM and is durable. Consequently, this instruction stalls the CPU for the entire duration of accessing the PM, which is expected to be expensive in terms of CPU cycles.

Durability can also be ensured by using asynchronous write-back instructions, such as `clflushopt` or `clwb` [120]. We adopt this approach in this chapter since it can be up to an order of magnitude faster than `clflush` [121]. Multiple invocations of these instructions are not ordered, so multiple cache lines can be flushed in parallel. Since these instructions do not stall the CPU and can be processed in the background, we consider the cost of invoking such instructions to be zero. Of course, this also means that invoking write-back asynchronous instructions is not sufficient to ensure durability.

In order to ensure that an asynchronous write-back completes and data is made durable, a *fence instruction* is required, which stalls the CPU until all active asynchronous write-back instructions complete. The fence instruction stalls the CPU for the entire duration for accessing the PM, which can be expensive. Thus, our focus in this work is reducing the number of such fences. We emphasize the fact that that it is possible to execute a fence while no asynchronous cache line flush instructions are active, in which case the CPU does not (necessarily) access the PM. We denote an execution of a fence while asynchronous cache line flushes are pending by *persistent fence*.

Additional details of the memory ordering model to PM are available in [64]. Briefly, two writes separated by a persistent fence are guaranteed to reach PM in order. However, standard fences (e.g., `mfences`) do not generally guarantee ordering to the PM, unless all writes address the same cache line.

4.2.2 Processes and Operations

We consider a set of n processes that communicate through shared memory access primitives. We make no assumptions on the relative speeds of the processes; in particular, at any point in time, processes may be delayed for arbitrary or even infinite amounts of time. As in previous work [127], we also assume that the whole system can crash at any point in time and potentially recover later. On such a full-system crash, we assume that the contents of PM are preserved but that the contents of the processor's registers and caches are lost. Processes running at the time of the full-system crash also crash and are replaced by new processes after recovery. After a system recovers and before resuming normal operation, we assume that a (potentially empty) *recovery* routine is invoked in order to bring the persistent objects on the PM back to a consistent state.

We classify operations on an object as *read-only* or *update*. Updates are operations that influence the result of subsequent operations; read-only operations do not influence later operations. Updates also read the state of the object and have return values. The *state* of the object is the sequence of update operations applied to the object; the first operation must be INITIALIZE. This definition implies that update operations are deterministic: applying the same sequence of updates on the object always results in the same state.

Our universal construction assumes the existence of a *compute* method. Given a read operation r and the state of the object s (i.e., the sequence of update operations on the object), this method computes the returned value of r applied to s. For an update operation u, the returned value is computed on the state of the object s immediately after appending u.

4.3 ONLL: a Primer

We give in this section a high-level view of ONLL, our universal construction that takes any deterministic object O and produces a lock-free durably linearizable implementation of O that requires at most one persistent fence per update operation and no persistent fence for read-only operations. Broadly, the execution of an operation under ONLL follows three stages: (1) *order* (in which the linearization order of the operations is established), (2) *persist* (in which the operation is made persistent) and (3) *linearize* (in which the operation is linearized)[1]. We first present the rationale behind these three stages and then give an overview of how ONLL works.

[1] Formally, the linearization point of an operation can be reasoned about only after the entire event history is known. We say that the linearization point of an operation is at time t if for any valid sequence of events after time t the linearization point of the operation can be set to time t.

We end the section with a concrete example, illustrating a shared counter implementation produced by ONLL.

4.3.1 Rationale

Not performing any persistent fences when reading is a highly desirable property. However, this imposes some constraints on the design of ONLL:

1. The linearization point of an update operation cannot coincide with the time when the operation reaches PM. This is because PM can only be written using simple writes (as opposed to the cache that supports CAS), so it cannot in general serve as a synchronization point. A reader cannot distinguish whether data already reached PM or not.
2. The linearization order of update operations must be known before the operation is written to PM. This is because PM must contain enough information to replay this information in the correct (linearization) order.
3. The linearization point of an update operation must happen after the write to PM.

The last constraint was derived by the following contradiction. Suppose that the linearization point of an update does not happen after the write to the PM. Then, a reader may observe the update operation before it reaches PM. There are three cases, each leading to a different contradiction:

- The reader finishes its operation before the dependent update is persisted. This breaks linearizability if the reader performs an external operation (e.g., print) before the dependent update reaches PM and the system crashes afterward. It is not possible to recover the update, but the dependent read was already observed.
- The reader waits for the dependent update to be persisted. This breaks lock-freedom since the process executing the update operation may stall arbitrarily long.
- The reader helps the dependent update to persist. This breaks the property of never executing a persistent fence for read-only operations.

4.3.2 ONLL Design

The design of ONLL derives naturally from the above-mentioned constraints. Since the linearization order of an update operation must be known before the operation is made durable, and the operation must be made durable before it is linearized, an update operation u under ONLL has three stages: order, persist, linearize.

First, in the order stage, a descriptor d is created for u and is appended to the tail of a shared *execution trace*. Second, in the persist stage, u and operations preceding it that are not

yet persistent are appended to a per-process persistent log (residing in PM), along with the ordering information. Third, in the linearize stage, u becomes visible to other operations when the available flag of u, or a later operation, is set. Finally, the returned value of u is computed based on the state of the object according to the execution trace up to d. So u is linearized at the linearize stage, after the write to PM, but according to the order computed at the order stage.

For a read-only operation r, the execution trace is traversed, starting from the tail, until the first descriptor d_{first} with a set available flag is reached. The return value of r is then computed based on the object state up to d_{first}, as recorded in the execution trace.

This design ensures that the linearization point of an operation happens after the write to the PM, so that readers are never required to wait or help with persisting the update operation. Moreover, after a crash, the PM contains enough information to recover the state of the object in same order as the linearization points.

4.3.3 Illustration: Shared Counter

We illustrate how our ONLL universal construction works for a concrete shared object: a counter. A counter holds an integer value and has two operations: *increment* and *read*. The first is an update operation that increments the counter's value and returns the new value. Read is a read-only operation that returns the value of the counter. In what follows, we walk through several increasingly complex executions of the counter (shown in Figure 4.1), to illustrate various situations that can arise with our construction.

Sequential update and read. In the first execution, a single process p_1 executes an update operation (increment), followed by a read-only operation. Initially, both the execution trace and p_1's persistent log is empty. Process p_1 creates a new node n with execution index equal to 1 and available flag unset. Then, p_1 appends to the persistent log an entry containing all operations that have not been persisted yet (just n in this case). To finalize the update, p_1 sets n's available flag and returns the new value of the counter, 1. Next, p_1 performs a read operation by traversing the execution index from tail to head, stopping at the first node with a set available flag. In our case, there is only one node n, and its available flag is set. The read thus computes its return value based on the state of the counter at n. n corresponds to a state in which one increment has been performed, so the read returns 1.

Update concurrent with reads. In the second execution, process p_1 is executing an update concurrently with two readers r_1 and r_2. The counter initially has value 1: there is already a node n_1 in the execution trace. The update appends a new node n_2 to the execution trace, appends the relevant entry to p_1's persistent log, and then pauses. r_1 traverses the execution trace from tail to head, stopping at n_1, the first node with a set available flag. p_1 resumes

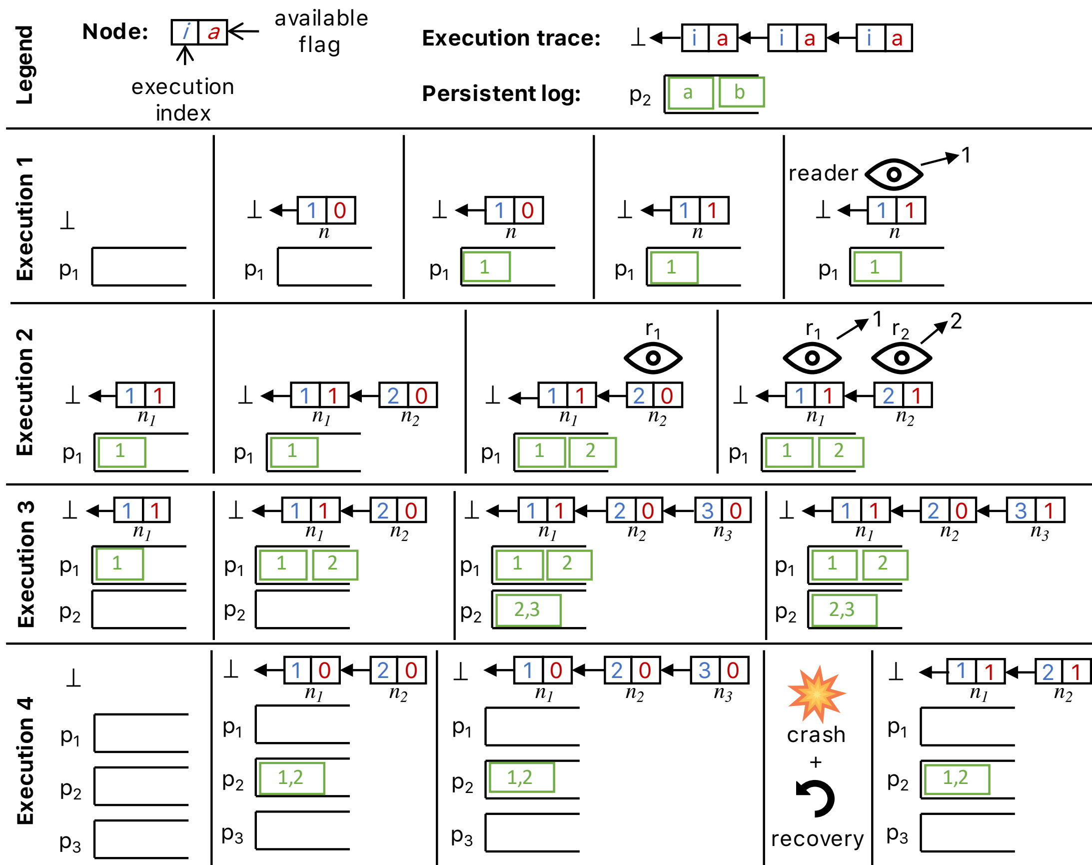


Figure 4.1 – Executions of a counter implemented using ONLL.

execution and sets the available flag of n_2. r_2 begins traversing the execution trace and stops at n_2. Finally, the three operations return: r_1 returns 1, based on the state of the counter at n_1; r_2 returns 2, based on the state at n_2; p_1's update returns 2, also based on the state at n_2.

Update helping another update. In the third execution, processes p_1 and p_2 are each executing an increment concurrently. Initially, the counter has value 1: the execution trace only contains node n_1. Process p_1 appends a node n_2 to the execution trace, adds the corresponding entry to the persistent log, and then pauses. p_2 also appends a node n_3 to the execution trace and then adds a persistent log entry containing all operations that have not been persisted yet: both p_1's update and p_2's update. Finally, p_2 sets the available flag of n_3 and returns 3. Any reader starting its traversal after n_3's available flag has been set will return 3, even though the available flag of n_2 has not yet been set.

Crash concurrent with updates and reads. In the fourth execution, processes p_1, p_2 and p_3 are each executing an update operation. Initially, the counter has value 0 and the execution trace is empty. Process p_1 appends a node n_1 to the execution trace and then pauses for the

rest of the execution. p_2 appends an execution trace node n_2, adds an entry to the persistent log corresponding to its own update and to the update of p_1, and then pauses without setting the available flag of n_2. p_3 also appends an execution trace node n_3, and starts adding a node to the persistent log corresponding to the updates of p_1, p_2 and p_3. The system crashes before any of the operations have returned.

After the crash, the state of the counter reflects the updates of p_1 and p_2. These can be reconstructed from p_2's persistent log, even though no available flag was set during the execution. The post-crash state of the counter does not however reflect p_3's update, because p_3 did not finish adding its persistent log entry.

Since no available flag was set during the execution, any reader concurrent with the updates will return 0, the initial value of the counter. Post-crash readers will return 2.

4.4 ONLL: a Universal Construction

In this section, we first detail the data structures required by ONLL and then we describe the ONLL algorithm itself.

4.4.1 Data Structures

The ONLL algorithm depends on two basic building blocks: a single-fence persistent log and a lock-free queue.

We assume that an update operation can be stored in (persistent) memory by using an `operation` structure; input parameters are considered part of the operation and are thus also reflected in the operation structure.

4.4.1.1 Persistent Log Usage

ONLL uses per-process persistent logs. We leverage the log implementation of Cohen et al. [64], which uses only one persistent fence per append. Each append invocation records up to *MAX-PROCESSES* operations, the number of recorded operations, and an execution index; pseudo code is provided in Listing 4.1. The first operation in the operation array is the current update operation executed by the process. The rest of the operations in the operation array are used to help other processes to persist their data. The `executionIndex` is a unique index that represents the ordering of the linearization point of the first operation. Operations in the array are sequential, so that the execution index of the k-th help operation is `executionIndex` $- k$.

Listing 4.1 – The recordEntry structure used by ONLL.

```
struct recordEntry{
    operation ops[MAX_PROCESSES];
    int num_ops; //between 1 and MAX_PROCESSES
    long executionIndex;
}
```

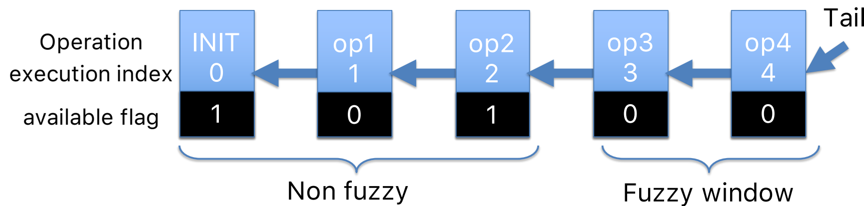


Figure 4.2 – The execution trace and the fuzzy window. Since op2 has a set available flag, all operations preceding it, including op1, are considered part of the non-fuzzy window. op3 and op4 have no later operation with a set available flag and so are the fuzzy window of the execution trace.

4.4.1.2 Transient Execution Trace

The second data structure used by ONLL is a *transient* (i.e., not necessarily stored in PM) execution trace of the object. This represents the sequence of all update operations applied to the object. Recall that the execution trace is equivalent to the state of the object. We emphasize that read-only operations do not appear in the execution trace of an object since they do not influence the state of the object; in our design, a read-only operation never writes to shared memory or to PM.

The sequence of the update operations in the execution trace is partitioned into a *non-fuzzy* prefix and a *fuzzy window* postfix. The fuzzy window represents a set of currently executing operations that are not yet guaranteed to reside on PM and their linearization point has not yet occurred. The non-fuzzy prefix consists of all other operations, which are guaranteed to reside on PM and their linearization point has already occured. The fuzzy window is implemented by assigning an *available* flag for each operation in the execution trace. The fuzzy window spans from from the latest operation in the execution trace up to (but not including) the latest operation with available flag set. Available flags can be set in any order, depending on the speed of the relevant process. A set available flag is never cleared.

It is important to note that the fuzzy window is continuous: if an operation *op* has its available flag unset, but a later operation has its available flag set, then *op* is not part of the fuzzy window. An illustration appears in Figure 4.2.

The execution trace is implemented in a lock-free manner, based on a lock-free queue algorithm [180]. A slight difference from a traditional lock-free algorithm is the need to compute

the *execution index* of each operation, which counts the number of update operations in the execution trace before the current operation. Pseudo code for the queue operation appears in Algorithm 4.2. Computing the execution index of a new operation is done at Line 34.

The execution trace data structure supports the `latestAvailable` method for reading the latest operation with its available flag set (the latest operation in the non-fuzzy part of the execution trace). Pseudo code appears at Algorithm 4.2 Line 38. It is important to note that the `latestAvailable` method returns the latest *observed* available operation, which might not be the actual latest operation in the non-fuzzy part of the execution trace. This is because it is possible that while the `latestAvailable` method is traversing the operations with available flag unset, the availability of a later operation is set. In fact, it is possible that the returned node was never the latest operation in the non-fuzzy part. ONLL is correct despite this anomaly, as is described later (Proposition 4.5.9).

4.4.2 ONLL Algorithm

Our algorithms for updating the object and reading the object are presented in Algorithms 4.3 and 4.4, respectively.

An update operation starts by adding a new node to the execution trace at Line 44. This corresponds to setting the linearization order of the update operation without making it visible to read-only operations and without linearizing it[2]. At Line 45, the fuzzy window of the operation is computed. This corresponds to the set of operations preceding the current operation but are not yet guaranteed to be persistent. (We later show — Proposition 4.5.2 — that this computation is finite since there are at most *MAX-PROCESSES* nodes in the fuzzy window). Helping these operations to persist on the PM prevents waiting for an unresponsive process. Then, the current operation and the helped operations are persisted by appending them to the private, persistent log (Line 46). The update part finishes by writing the available flag. This corresponds to the linearization point of the operation (unless another process helped it[3]) and makes the operation visible to read-only operations. Finally, if the update operation also returns a value, this value is computed and returned to the caller.

A read-only operation gets the latest node with available flag set. This node corresponds to the latest node in the non-fuzzy prefix of the object state. Then, the return value is computed based on this state and returned to the caller.

After a system crash, the transient execution trace is reconstructed from the persistent logs of all processes.

The recovery process starts by adding the initialization operation to the execution trace, which

[2]We note that adding a new node uses a CAS instruction, which serves as a (concurrency) fence. However, no writes to the PM are pending, so this fence does not count as a persistent fence.

[3]Helping is done by setting the available flag of a later operation, which logically linearizes the current operation. There is no help for setting the available flag: it is set only by the process executing the operation.

Algorithm 4.2 – The execution trace used by ONLL.

```
1  struct queueNode{
2      operation op;
3      long idx;
4      atomic<bool> available;
5      queueNode *next;
6      set<operation> getFuzzyOps(){
7          queueNode *curr=this;
8          set<operation> ops;
9          while(curr->available==false){
10             ops.add(curr->op);
11             curr=curr->next;
12         }
13         return ops;
14     }
15     queueNode *latestAvailable(){
16         queueNode *curr=this;
17         while(curr->available==false){
18             curr=curr->next;
19         }
20         return curr;
21 }   }

23 class executionTrace{
24     atomic<queueNode *> tail;
25     executionTrace(){
26         tail=new operation(INITIALIZE, 0, true, null);
27         //also serves as a sentinel
28     }
29     void insert(queueNode *node){
30         queueNode *ltail;
31         node->available.store(false, memory_order_relaxed);
32         do{
33             ltail = tail;
34             node->idx = ltail->idx+1;
35             node->next = tail;
36         }while(tail.compare_exchange_weak(ltail, node))==false);
37     }
38     queueNode *latestAvailable(){
39         queueNode *curr=tail;
40         return curr->latestAvailable();
41 }   }
```

Algorithm 4.3 – The update operation in ONLL.

```
42 Update(operation op){
43     queueNode *node = new queueNode(op);
44     executionTrace.insert(node);
45     operation fuzzyOps[MAX_PROCESSES] = node->getFuzzyOps();
46     persistentLog.append(fuzzyOps, node->idx);
47     node->available.store(true, memory_order_seq_cst);
48     return compute(node, op);
49 }
```

Algorithm 4.4 – The read operation in ONLL.

```
50 Read(operation op){
51     queueNode *node = executionTrace.latestAvailable();
52     return compute(node, op);
53 }
```

Algorithm 4.5 – Recovery in ONLL.

```
54 executionTrace.insert(queueNode(INITIALIZE)).setAvailable();
55 for(i=1; true; i++){
56     Find log entry E with lowest execution index j:j≥i.
57     if(E does not exist)
58         break;
59     operation op=E.ops[j-i];
60     executionTrace.insert(queueNode(op)).setAvailable();
61 }
```

serves as a sentinel node in the execution trace. Then, it iteratively searches for the next operation in the execution trace by looking into the persistent logs of all processes. If the operation op was not stored into any persistent log, the recovery process looks for an operation with a higher execution index and finds op by looking into the helped operations by the later operation. Finally, the found operation is pushed to the execution trace and the available flag is set. Recovery code is illustrated in Algorithm 4.5.

4.5 ONLL: Correctness

In this section, we prove the following theorem.

Theorem 4.5.1. *For any deterministic object O, there exists a lock-free durably linearizable implementation of O that requires at most one persistent fence per update operation and no persistent fence per read-only operation.*

We prove the theorem by first showing that ONLL is lock-free and then that it is durably linearizable.

4.5.1 Lock-freedom

An implementation is *lock-free* if it guarantees that infinitely often, some operation returns in a finite number of steps. More specifically, if any process is permanently taking steps, some operation will eventually return.

The lock-freedom proof uses the following proposition, showing that traversing the size of a fuzzy window is bounded regardless of the initial node.

Proposition 4.5.2. *At any time during any execution of* ONLL *and any MAX-PROCESSES+1 consecutive nodes in the execution trace, at least one has an available flag that is set.*

Proof. Let t be any time during the execution and let $S = \{n_i, \ldots, n_{i+MAX-PROCESSES}\}$ be *MAX-PROCESSES*+1 consecutive nodes in the execution trace. Clearly, there are at least two nodes $n_j, n_k \in S$ that correspond to two operations by the same process.

In our model, two operations cannot be executed by the same process at the same time: the first operation must return before the second operation can be invoked. Thus, let n_j be the earlier operation and the t' be the time it finished. Clearly, $t' < t$ since at time t the operation corresponding to n_k already appended itself to the execution trace, implying that it was already invoked.

According to Algorithm 4.3, an operation does not finish before setting the available flag and executing a memory fence. Thus, at least n_j has a set available flag, as required. □

Recall that a set available flag is never unset; this property together with Proposition 4.5.2 imply that `getFuzzyOps` and `latestAvailable` are wait-free: they cannot be interfered with by other processes and they always finish in O(*MAX-PROCESSES*) steps.

Lemma 4.5.3. *Suppose that compute — the function for computing the return value of an operation — always finishes in a finite time. Then* ONLL *is lock-free.*

Proof. Reads first find the `latestAvailable` node and then execute *compute* on the resulting node. `latestAvailable` finishes in a bounded number of steps and is thus wait-free. *Compute* operates on a prefix of the execution trace starting with the `latestAvailable` node. This prefix is never modified by any process (except for the available flag, which is ignored by *compute*). Since we assume *compute* finishes in a bounded number of steps, it is also wait-free. Thus, reads are wait-free.

Next we consider updates. Appending to the execution trace is a lock-free operation. Getting the fuzzy window of an operation is wait-free since it always finishes in a bounded time. Appending to the process' private persistent log is also wait-free since an append is never interrupted by other processes and it finishes in a bounded number of steps. Finally, setting the available flag of a node is a wait-free operation. Thus, we conclude that updates are lock-free.

□

4.5.2 Durable linearizability

We first recall the concept of durable linearizability and then proceed with the proof of durable linearizability in ONLL.

4.5.2.1 Technical Preliminaries

The execution of a concurrent system is modeled by a *history*, a sequence of events. Events can be operation *invocations* and *responses*. Each event is labeled with the process and with the object to which it pertains. A *subhistory* of a history H is a subsequence of the events in H.

A response *matches* an invocation if they are performed by the same process on the same object. An operation in a history H consists of an invocation and the next matching response. An invocation is *pending* in H if no matching response follows it in H. An *extension* of H is a history obtained by appending responses to zero or more pending invocations in H. $complete(H)$ denotes the subhistory of H containing all matching invocations and responses.

For a process p, the *process subhistory* $H|p$ is the subhistory of H containing all events labeled with p. The object subhistory $H|O$ is similarly defined for an object O. Two histories H and H' are equivalent if for every process p, $H|p = H'|p$.

A history H is *sequential* if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. A history is *well-formed* if each process subhistory is sequential.

A *sequential specification* of an object O is a set of sequential histories called *legal* histories of O. A sequential history H is *legal* if for each object O appearing in H, $H|O$ is legal.

An operation op_1 *precedes* op_2 in H (denoted $op_1 \rightarrow_H op_2$) if op_1's response event appears in H before op_2's invocation event. Precedence defines a partial order on the operations of H.

Definition 4.5.4 (Linearizability)**.** *A history H is* linearizable *if H has an extension H' and there is a legal sequential subhistory S such that*

- ***L1*** *$complete(H')$ is equivalent to S*
- ***L2*** *if an operation op_1 precedes an operation op_2 in H, then the same holds in S.*

Informally, this definition is equivalent to saying that an object is linearizable if every operation appears to take effect instantaneously at some point (the *linearization point*) between invocation and response. Incomplete operations (invocations without matching responses) may or may not have a linearization point.

Durable linearizability [127] captures the fact that an object's state should remain consistent even across crashes and recoveries, without "erasing" any completed operations.

Definition 4.5.5 (Consistent cut)**.** *Given a history H, a consistent cut of H is a subhistory P of H such that if $op_2 \in P$ and $op_1 \rightarrow_H op_2$, then $op_1 \in P$ and $op_1 \rightarrow_P op_2$.*

Definition 4.5.6 (Durable linearizability)**.** *An object O is* durably linearizable *if its states immediately before and immediately after a crash and recovery reflect histories H and H' respectively*

such that (1) H and H' are linearizable and (2) H' is a consistent cut of H for which every complete operation op in H is also in H'.

In other words, all operations that completed before the crash must be included in H', but some operations that had not yet completed may be excluded and thus not be reflected in the post-recovery state of the object. However, the operations in H' must constitute a consistent cut of H, meaning that if some operation op is included in H', then so must all operations on which op depends.

Informally, an object O is durably linearizable if, in any history H produced by O, every operation appears to take effect instantaneously at some point (the *linearization point*) between invocation and response. Incomplete operations (invocations without matching responses, either due to delayed processes or to system crashes) may or may not have linearization points. Operations concurrent with a crash may be reflected in the post-crash state of the object (in which case these operations have linearization points before the crash) or may not be reflected (in which case these operations have no linearization points).

4.5.2.2 Durable Linearizability in ONLL

Lemma 4.5.7. ONLL *is durably linearizable.*

The proof of durably linearizable for ONLL proceeds in 4 steps. First, we define linearization points for all update operations and define a time point for a crash (if a crash occurs). Second, we prove that the linearization point of an update falls between its invocation and response. Third, we show that a read is linearizable. Fourth, we show that the state of the object after crash and recovery matches the state of the object before recovery, according to the update operations linearized before the crash.

We start by defining a point in time where each update operation linearizes. When defining linearization points, we use the following convention. An integral time, denoted by $t_\Box$, corresponds to a specific instruction executed on the durable shared object; a non-integral (i.e., fractional) time, denoted by $t_\Box - a \cdot \epsilon$ denotes a linearization point that happens before time $t_\Box$ but does not relate to a specific event during execution. We assume ϵ is sufficiently small and a is a positive finite number so that $t_\Box - 1 < t_\Box - a \cdot \epsilon < t_\Box$.

The linearization point of an update operation op with execution index i is the earlier between (1) the time t_i the i-th available flag was set at the end of op or (2) immediately before the time t_j when the j-th available flag was set for $j > i$. To avoid distinguishing between these two cases, we consider the first j-th available flag that was set such that $j \geq i$. The linearization point of op_i is $t_i = t_j - (j - i) \cdot \epsilon$ for a sufficiently small $\epsilon > 0$.

If a crash happens, let t_{crash} be the time of the crash; we assume this time is higher by at least one than the last instruction executed before the crash. Clearly, operations that returned

before the crash are included in the post-crash state of the object. We now examine operations that were ongoing at t_{crash}. Let op_i be such an ongoing operation, with execution index i. We examine several cases:

1. Some operation $op_j : j \geq i$ persisted op_i and op_j set its available flag at time t_j. Then op_i is linearized at time $t_i = t_j - (j - i) \cdot \epsilon$ as discussed above.
2. op_i either (a) persisted itself, but did not set its flag, or (b) was persisted by one or more other operations (Algorithm 4.3 Line 46), but none of these operations set their flag. To establish the linearization point of op_i, let op_l be the operation with highest execution index ($= l$) that finished persisting before t_{crash}. op_i is linearized at $t_i = t_{\text{crash}} - (l - i) \cdot \epsilon$.
3. op_i was not persisted at all. That is, no operation $op_j : j \geq i$ finished appending to the persistent log at Algorithm 4.3 Line 46. Then, op_i is not linearized and is lost in the crash.

Proposition 4.5.8. *The linearization point of an update operation falls between the invocation and the response of this operation.*

Proof. Consider an update operation op_i and let $j \geq i$ be the first index such that the j-th available flag that was set. The linearization point of op_i is $t_j - (j - i) \cdot \epsilon$.

When op_i is inserted in the execution trace, the latest operation in the trace is $i - 1$; clearly, op_j is inserted in the execution trace no earlier than op_i was inserted in the execution trace (note that i can be equal to j, in which case the times are equal). Setting the j-th available flag is done at time t_j. Inserting the j-th node in the execution trace happens earlier and is related to the execution of an actual instruction; thus, its time is at most $t_j - 1$. Recall that ϵ is small enough so that $t_j - 1 < t_j - (j - i) \cdot \epsilon$. Thus, op_i is inserted in the execution trace before time $t_j - (j - i) \cdot \epsilon$. op_i is invoked before is it inserted in the execution trace, establishing that the linearization point of op_i happens after its invocation.

Operation i does not finish before setting the i-th available flag. Clearly, the i-th available flag is set no earlier than the first j-th available flag is set, $j \geq i$. By definition of the linearization points, setting the j-th available flag happens at time t_j. Thus, the response to op_i happens after time $t_j \geq t_j - (j - i) \cdot \epsilon$. □

Next we consider the linearization point of reads. If a reader were to traverse an atomic snapshot of the execution trace and found the last node in the non-fuzzy window, its linearization point would follow immediately from the definition of the linearization points of writes. But ONLL readers traverse the execution trace directly (and not an atomic snapshot thereof) and thus they may find a node corresponding to a concurrent update[4]. In this case, we show that the linearization point of the read can be set to immediately after the linearization point of the concurrent update operation.

[4] For more details see Appendix C

Proposition 4.5.9. *A read-only operation has a linearization point between the invocation and the response of the operation, such that the return value of the operation corresponds to the state of the object at the linearization point.*

Proof. A read-only operation traverses the execution trace from the tail until it reaches a node with a set available flag. The resulting node is the state on which the returned value is computed. Suppose that the tail pointed to op_j and the highest index operation with available flag set is $op_i : i \le j$. Consider the time t the read-only operation reads the tail (Line 7); either the i-th available flag was set at time t or not. If the i-th available flag was set at time t, then the linearization point of the read is set to time t. This clearly falls between the invocation and the response. The state of the object at time t contains op_i since its available flag is set, but not any operation in the range $[i+1, j]$ since their available flag is unset at time t. The latter is true since otherwise the traversal would find a later node $k \ge i+1$ with a set available flag.

Next, consider the case that the i-th available flag was not set at time t. Denote by t_e the time the read-only operation finds that the i-th available flag is set. At time t all operations in the range $[i, j]$ have unset available flag and at time t_e the i-th available flag is set. Thus, the linearization point of op_i falls between time t and t_e. We set the linearization time of the read-only operation to immediately after the linearization time of op_i and before the linearization point of any other update operation[4]. Since the linearization point is between time t and t_e, it is after the invocation of the read-only operation and before the response, as required. □

Proposition 4.5.10. *The state of the* ONLL *object after a crash includes all the operations that were linearized before the crash, executed in linearization order, and none of the operations that were not linearized before the crash.*

Proof. By definition of the linearization points before a crash, the state of the object just before t_{crash} corresponds to the last operation that was written to the persistent log. After a crash, the recovery reconstructs the execution trace by traversing all persistent logs. Thus, the last node in the execution trace after recovery is the last operation that was written to the persistent log. The order of operations follows the `executionIndex`, which is stored on the persistent logs. Thus, the order of operations is equal before the crash and after recovery. It remains to show that all operations appearing in the execution trace before the crash also appear after recovery.

Suppose, by a way of contradiction, that an operation i that appeared before the crash does not appear after the crash. Since the latest operation is the same, there exists an operation $op_j : j > i$ that appears both before the crash and after recovery. We pick the operation with smallest j (that is larger than i). Operations that have a set available flag clearly appear in the log. Thus, all operations in the range $[i, j-1]$ must have their available flag unset until t_{crash}. But since op_j persisted in the persistent log before t_{crash}, it must have been added to the execution trace before t_{crash}. According to Proposition 4.5.2, there are at most *MAX-*

PROCESSES - 1 operations between op_i and op_{j-1} since the available flag of op_j is unset when it is appended to the execution trace.

But op_j appended to the persistent log all operations in the range $[i, j]$, including op_i, so it appears in a persistent log. This contradicts our assumption that op_i is not on the persistent log. □

The proof of Lemma 4.5.7 follows from Propositions 4.5.8, Proposition 4.5.9 and Proposition 4.5.10. The proof of Theorem 4.5.1 follows from Lemma 4.5.3 and Lemma 4.5.7.

Interestingly, ONLL also provides detectable execution [91]. After recovery, it is possible to check if a given update operation appears in the execution trace. The operation was linearized before the crash if and only if it appears in the execution trace after recovery.

4.6 Lower Bound

We show that in any lock-free implementation of a durably linearizable object, there exists some execution in which every update operation must issue at least one persistent fence. This is trivially true if operations are executed sequentially (otherwise a crash immediately after an operation *op* would mean that *op* is not reflected in the state of the object after recovery). Intuitively, the need for processes to persist their operations also manifests in some concurrent executions: if some process p were to always rely on other processes to persist its operations, then p might need to wait indefinitely if those other processes are delayed, thus violating lock-freedom.

We prove this intuition below, through several intermediate results, after defining relevant terminology.

Terminology. We say that a process p *runs solo* between events A and B in an execution if p is the only process taking steps between A and B in that execution. Two sequences of operations H_1 and H_2 are *equivalent* (denoted $H_1 \equiv H_2$) if any possible execution, when started from H_1 produces the same results (operation return values) as when started from H_2. We denote by $H_1 \cdot H_2$ the sequence obtained by executing sequence H_2 after sequence H_1. An operation op is an *update* if there exists a sequence H such that $H \cdot op \not\equiv H$. For ease of description, we define a *state* as a sequence of update operations, starting with INITIALIZE.

Lemma 4.6.1. *Let A, A' and B be sequences of operations such that $A \cdot B \not\equiv A' \cdot B$. Then $A \not\equiv A'$.*

Proof. Assume that $A \equiv A'$. Then, by definition of equivalence, $A \cdot B \equiv A' \cdot B$, a contradiction. □

Lemma 4.6.2. *Let op be any update and H be any state such that $H \cdot op \not\equiv H$. Then, if for some $n \geq 2$, $H \cdot op^{n-1} \equiv H \cdot op^n$, the following property holds: $\forall j \geq 1, H \not\equiv H \cdot op^j$.*

Proof. Assume by contradiction that $\exists j \geq 1 : H \equiv H \cdot op^j$. Then for all $k \in \mathbb{N}$, $H \cdot op^{kj} \equiv H$. Then (1) $H \cdot op^{n-1} \cdot op^{\lceil \frac{n}{j} \rceil j-(n-1)} \equiv H$ and (2) $H \cdot op^n \cdot op^{\lceil \frac{n}{j} \rceil j-(n-1)} \equiv H \cdot op$. The left sides of (1) and (2) are equivalent (because $H \cdot op^{n-1} \equiv H \cdot op^n$), but the right sides are not equivalent. We have reached a contradiction. □

Theorem 4.6.3. *In a n-process system, for any lock-free durably linearizable implementation of an update operation op, there is an execution in which (1) all processes call op concurrently and (2) each process performs at least one persistent fence during its call to op.*

Proof. By definition of an update operation, there exists a state H such that op applied from H produces a different state $H \cdot op \not\equiv H$.

We now consider two cases: (1) $H \cdot op^{n-1} \not\equiv H \cdot op^n$ and (2) $H \cdot op^{n-1} \equiv H \cdot op^n$. For each case, we construct an execution in which the n processes $p_1, ..., p_n$ call op concurrently and all necessarily perform persistent fences.

Case 1: $H \cdot op^{n-1} \not\equiv H \cdot op^n$. We construct the following execution:

- Starting from H, let p_1 call op and run solo until just before the response of op. p_1 will eventually reach this point, due to the lock-freedom of the implementation. p_1 will perform at least one persistent fence before being preempted. Otherwise, let p_1 resume and perform the very next step of returning from op; if a crash occurs after this response, after recovery the contents of persistent memory will be identical to that at H, which is inconsistent with the only possible linearization $H \cdot op \not\equiv H$.
- Let p_2 call op and run solo until just before op returns (p_2 eventually returns due to lock-freedom). p_2 performs at least one persistent fence during its call to op. Otherwise, let p_2 return from op and let p_1 resume and perform the step of returning from op. If a crash occurs immediately after, at recovery the contents of memory will be identical to $H \cdot op \not\equiv H$, but the only possible linearization is $H \cdot op \cdot op$. By Lemma 4.6.1, $H \cdot op \cdot op \not\equiv H \cdot op$ (taking $A = H \cdot op$, $A' = H \cdot op \cdot op$ and $B = op^{n-2}$). We have reached a situation in which the only possible linearization is not compatible with the contents of memory, a contradiction; thus p_2 does indeed perform at least one persistent fence during its call to op.
- Continue with $p_3, ..., p_n$, each time calling op and preempting the process just before returning. As with p_2, each process will perform at least one persistent fence before being preempted.

Case 2: $H \cdot op^{n-1} \equiv H \cdot op^n$. We construct the following execution:

- Starting from H, let p_1 call op and run solo. If left to run solo long enough, p_1 will eventually perform a persistent fence. Otherwise, p_1 either never returns from op,

violating lock-freedom, or returns from op without performing a persistent fence. In the latter case, a crash may occur immediately after the response of op; upon recovery, the contents of persistent memory will be identical to those at H, which is inconsistent with the fact that $H \cdot op \not\equiv H$.

- Preempt p_1 just before the first persistent fence.
- Let p_2 call op and run solo. If left to run solo long enough, p_2 will eventually perform a fence. Otherwise, if p_2 returns without a fence and the system crashes afterwards, the contents of persistent memory are identical to H, which is inconsistent with the possible linearizations $H \cdot op$ (i.e., p_1's call is not linearized and p_2 call is linearlized) and $H \cdot op \cdot op$ (i.e., both p_1's call and p_2's call are linearized); this is due to Lemma 4.6.2 ($H \not\equiv H \cdot op$, $H \not\equiv H \cdot op \cdot op$).
- Continue in this way with processes $p_3, ..p_n$.
- For each process $p_n, ...p_1$, resume the process for one step—the persistent fence it was about to perform—then preempt it and move to the next process.

□

Our lower bound result holds for detectable execution [91] as well, since it is a stronger criterion than durable linearizability (an implementation satisfying the former requires at least as many persistent fences as an implementation satisfying the latter).

4.7 Related Work

Safety criteria. Several safety criteria have been proposed in the crash-recovery model. Persistent atomicity [100] requires any operation interrupted by a crash to be linearized or aborted before any later invocation by the pending process. In the same situation, recoverable linearizability [29] requires the operation to be linearized or aborted before any later invocation by the pending process on the same object. These two criteria assume that processes may crash and recover independently. However, it can be argued that this model is unnecessarily general, since processes typically crash together in a full-system crash (e.g., restart). In a more restricted model that only allows such full-system crashes, the two criteria become indistinguishable, and equivalent to durable linearizability [127], the safety condition adopted in this chapter.

Work has been done on verifying linearizability for traditional transient objects [95, 157, 212]. It would be interesting to see if such verification techniques could be extended to verify durable linearizability for persistent objects.

Our upper bound algorithm relates in an interesting way to Section 4.1 of Izraelevitz et al. [127]. In that section, the authors state that the linearization point of an operation must happen before it is persisted. However, our ONLL construction linearizes an operation *op after* the time

when *op* persisted and generates the linearization order before persisting, so that it knows what to persist. In fact, as discussed in Section 4.3, we argue that having the persist point earlier than the linearization point is necessary for any lock-free algorithm where readers never execute a persistent fence.

The same section of [127] contains a theorem (Theorem 2), which provides a set of sufficient conditions for an object to be durably linearizable. While at first glance it may seem that our ONLL algorithm contradicts this theorem, this is not the case: ONLL is durably linearizable, without satisfying the condition that operations are linearized before they are persisted.

General transformations. Related to the generality of our upper bound construction, there has been work [52, 56, 118] on generating correct persistent applications from existing code (designed for DRAM). However, in contrast to our work, these approaches assume the application is already multi-threaded and generally also assume lock-based code. Moreover, the focus in this work [52, 56, 118] is on lessening the programming effort necessary to transform applications, not on achieving optimality in terms of the number of persistent fences used.

In the same vein of generality, our work shares similarities with the universal construction of Herlihy [109, 110]. In both cases, the construction yields a correct concurrent implementation of an object from its sequential specification.

Transactions. Significant work has been done on transactions as a means of interacting with PM [33, 55, 63, 75, 96, 126, 138, 141, 163, 175, 224]. These efforts share similarities to our work in the following sense: they strive for generality, they aim to reduce the cost of interacting with PM, and they often use logging. Yet, these works do not consider lock-freedom as a progress guarantee. Also, whereas in transactions logging is used to help maintain the consistency of application state, in our construction, the log *is* the state.

Persistent data structures. A specific class of shared objects are concurrent data structures [115]. There has been some work on designing efficient data structures for PM, focused mostly on indexing trees [57, 153, 189, 229]. This is natural, given that indexing trees are used extensively in data structures and file systems. Recently, Friedman et al. [91] have proposed three lock-free durable queue algorithms. These are specific approaches, not easily generalizable to other data structures or to other shared objects.

Lower bounds. Related to our lower bound result, Attiya et al. [21] have shown that linearizable implementations of strongly non-commutative operations cannot completely eliminate the use of expensive synchronization primitives such as memory barriers and atomic instructions (whose effects also include the effects of memory fences). This seems to imply that any implementation of a durably linearizable update operation requires (at least in some executions) two fences: one to account for the cited lower bound and one to account for

our lower bound. However, since the effects of a memory fence also include stalling until pending cache line flushes have completed, a memory fence can also count as a persistent fence if flushes to PM are pending. Thus, it might be possible in some cases to implement a persistent object using only one (memory) fence per update operation, accounting for both our lower bound result and that of Attiya et al. We leave open the questions of when and if such one-fence updates are indeed achievable.

4.8 Concluding Remarks

In the PM era, programmers will need persistent and concurrent data structures. The performance of these is strongly influenced by the number of persistent fences executed for each operation. This chapter shows that lock-free implementations require exactly one persistent fence for any update operation to ensure correctness. Our upper bound uses a novel ordering scheme to persist operations before their linearization points. Our lower bound captures the very fact that processes cannot rely on each other to persist updates and thus shows that one cannot hope to reduce the number of persistent fences while still guaranteeing durable linearizability and lock-freedom.

Below, we discuss possible extensions of our results and future directions left open by our work.

Wait-freedom. According to the proof of Lemma 4.5.3, the only operation in our ONLL construction that is not wait-free is the execution trace transient data structure. Since this data structure is transient, standard techniques such as the wait-free construction of Timnat and Petrank [221] can be used to derive a wait-free execution trace data structure. Alternatively, a wait-free execution trace can be based on the wait-free queue of Kogan et al. [139]. ONLL can thus easily be made wait-free.

Compressing the execution trace. An ONLL object stores its state as a sequence of all operations applied to this object. This representation could be improved for specific cases, as most practical objects have an object-specific representation of their state. For example, for the shared persistent counter discussed in Section 4.3, an object-specific representation would be an integer field corresponding to the current value of the counter.

If such a representation exists, one could consider a hybrid approach that combines a small ONLL execution trace for correctness with an object-specific representation for efficiency. As explained below, this approach would have the double benefit of (1) allowing better read performance and (2) enabling memory reclamation, thus reducing memory consumption.

Readers in ONLL traverse the entire execution trace; thus, reading an object's state implies traversing all update operations in the history of the object. ONLL read performance can

be significantly improved by storing a local view per process, similarly to log-based systems [24, 25, 48]. The local view of process p includes (1) a representation r_p of the object up to some operation op_p and (2) the execution index of op.

A read by process p begins as before by finding the first execution trace node n with a set available flag. Then, p applies to its local representation r_p all updates between op_p and n. Then, p updates its local execution index to that of n. Finally, the read is served directly from r_p.

In this way, the overhead of a read is the difference between the execution index of the local view and the execution index of the shared object, which is expected to be significantly smaller than the number of nodes in the execution trace.

Another effect of storing the entire execution trace in ONLL is the inability to reclaim memory. Both execution trace nodes and persistent log entries have to be kept forever. If the state can be stored as a small object-specific representation, however, then there is no need to remember all update operations and log entries, thus significantly reducing memory consumption.

Execution trace nodes can be reclaimed if we use the following scheme. As before, each process p has a local transient representation of the object r_p, which p brings up to date periodically. Note that once a process p has applied an operation op from the execution trace to r_p, p will never need to read op again. Thus, once all processes have updated their local representations past op, the execution trace prefix up to op can be safely reclaimed.

We can go one step further and also reclaim persistent log entries. Each process p periodically records its local representation r_p in its persistent log, along with the execution index n of op_p. Afterwards, p can reclaim the memory of all persistent log entries with execution indexes smaller than n.

Lock-based implementations. At first glance, it might seem that by allowing implementations of persistent objects to be blocking, one could reduce the number of persistent fence instructions. For instance, the work of Cohen et al. [64] enables an implementation in which each process announces its operation and one of the processes applies all announced operations (similarly to flat combining [108]) using a single persistent fence. This implementation might seem to use only one persistent fence for every batch of concurrent operations. However, upon closer inspection, it is easy to realize that all pending operations pay the price of a persistent fence (by waiting while the combiner performs the fence), even without actually performing the fence.

5 Efficient Multi-Word Compare-and-Swap

In the previous chapter, we provided a better understanding of the fundamental costs associated with lock-free programming for persistent memory. In this chapter, we turn our attention to atomic multi-word compare-and-swap (MCAS), a powerful building-block for designing lock-free algorithms for persistent, as well as for volatile, memory. Despite its versatility, the widespread usage of MCAS has been limited because lock-free implementations of this primitive make heavy use of expensive compare-and-swap (CAS) instructions. Existing MCAS implementations indeed use at least $2k+1$ CASes per k-CAS. This leads to the natural desire to minimize the number of CASes required to implement MCAS.

We first prove in this chapter that it is impossible to "pack" the information required to perform a k-word CAS (k-CAS) in less than k locations to be CASed. Then we present the first algorithm that requires $k+1$ CASes per call to k-CAS in the common uncontended case. We implement our algorithm and show that it outperforms a state-of-the-art baseline in a variety of benchmarks in most considered workloads. We also present a durably linearizable (persistent memory friendly) version of our MCAS algorithm using only 2 persistence fences per call, while still only requiring $k+1$ CASes per k-CAS.

5.1 Introduction

Compare-and-swap (CAS) is a foundational primitive used pervasively in concurrent algorithms on shared memory systems. In particular, it is used extensively in *lock-free* algorithms, which avoid the pitfalls of blocking synchronization (e.g., that employs locks) and typically deliver more scalable performance on multicore systems. CAS conditionally updates a memory word such that a new value is written if and only if the old value in that word matches some expected value. CAS has been shown to be universal, and thus can implement any shared object in a non-blocking manner [110]. This primitive (or the similar load-linked/store-conditional (LL/SC)) is nowadays provided by nearly every modern architecture.

CAS does have an inherent limitation: it operates on a single word. However, many concurrent

algorithms require atomic modification of multiple words, thus introducing significant complexity (and overheads) to get around the 1-word restriction of CAS [41, 76, 98, 99, 158, 184]. As a way to address the 1-word limitation, the research community suggested a natural extension of CAS to multiple words—an atomic multi-word compare-and-swap (MCAS). MCAS has been extensively investigated over the last two decades [11, 12, 76, 98, 99, 106, 110, 182, 214]. Arguably, this work partly led to the advent of the enormous wave of Transactional Memory (TM) research [103, 105, 114]. In fact, MCAS can be considered a special case of TM. While MCAS is not a silver bullet for concurrent programming [81, 113], the extensive body of literature demonstrates that the task of designing concurrent algorithms becomes much easier with MCAS. Not surprisingly, there has been a resurgence of interest in MCAS in the context of persistent memory, where the persistent variant of MCAS (PMCAS) serves as a building block for highly concurrent data structures, such as skip lists and B+-trees [17, 226], managed in persistent memory.

Existing lock-free MCAS constructions typically make heavy use of CAS instructions [11, 106, 182], requiring between 2 and 4 CASes per word modified by MCAS. That resulting cost is high: CASes may cost up to 3.2× times more cycles than simple load or store instructions [73]. Naturally, algorithm designers aim to minimize the number of CASes in their MCAS implementations.

Toward this goal, it may be tempting to try to "pack" the information needed to perform the MCAS in fewer than k memory words and perform CAS only on those words. We show in this chapter that this is impossible. While this result might not be surprising, the proof is not trivial, and is done in two steps. First, we show through a bivalency argument that lock-free MCAS calls with non-disjoint sets of arguments must perform CAS on non-disjoint sets of memory locations, or violate linearizability. Building on this first result, we then show that any lock-free, disjoint-access-parallel k-word MCAS implementation admits an execution in which some call to MCAS must perform CAS on at least k different locations. (Our impossibility result focuses on *disjoint-access-parallel* (DAP) algorithms, in which MCAS operations on disjoint sets of words do no interfere with each other. DAP is a desirable property of scalable concurrent algorithms [124].)

We also show, however, in the chapter that MCAS can be "efficient". We present the first MCAS algorithm that requires $k+1$ CAS instructions per call to k-CAS (in the common uncontended case). Furthermore, our construction has the desirable property that reads do not perform any writes to shared memory (unless they encounter an ongoing MCAS operation). This is to be contrasted with existing MCAS constructions (in which read operations do not write) that use at least $3k+1$ CASes per k-CAS. Furthermore, we extend our MCAS construction to work with persistent memory (PM). The extension does not change the number of CASes and requires only 2 persistence fences per call (in the common uncontended case), comparing favorably to the prior work that employs $5k+1$ CASes and $2k+1$ fences [226].

Most previous MCAS constructions follow a multi-phase approach to perform a k-CAS

operation op. In the first (*locking*) phase, op "locks" its designated memory locations one by one by replacing the current value in those locations with a pointer to a *descriptor* object. This descriptor contains all the information necessary to complete op by the invoking thread or (potentially) by a helper thread. In the second (*status-change*) phase, op changes a status flag in the descriptor to indicate successful (or unsuccessful) completion. In the third (*unlocking*) phase, op "unlocks" those designated memory locations, replacing pointers to its descriptor with new or old values, depending on whether op has succeeded or failed.

In order to obtain lower complexity, our algorithm makes two crucial observations concerning this unlocking phase. First, this phase can be deferred off the critical path with no impact on correctness. In our algorithm, once an MCAS operation completes, its descriptor is left in place until a later time. The unlocking is performed later, either by another MCAS operation locking the same memory location (and thus effectively eliminating the cost of unlocking for op) or during the memory reclamation of operation descriptors. (We describe a delayed memory reclamation scheme that employs epochs and amortizes the cost of reclamation across multiple operations.)

Our second, and perhaps more surprising, observation is that deferring the unlocking phase allows the *locking* phase to be implemented more efficiently. In order to avoid the ABA problem, many existing algorithms require extra complexity in the locking phase. For instance, the well-known Harris et al. [106] algorithm uses the atomic *restricted double-compare single-swap* (*RDCSS*) primitive (that requires at least 2 CASes per call) to conditionally lock a word, provided that the current operation was not completed by a helping thread. Naively performing the locking phase using CAS instead of RDCSS would make the Harris et al. algorithm prone to the ABA problem (we provide an example in Appendix D.1). However, in our algorithm, we get ABA prevention "for free" by using a memory reclamation mechanism to perform the unlocking phase, because such mechanisms already need to protect against ABA in order to reclaim memory safely.

Deferring the unlocking phase allows us to come up with an elegant and, arguably, simple MCAS construction. Prior work shows, however, that the correctness of an MCAS construction should not be taken for granted: for instance, Feldman et al. [88] and Cepeda et al. [51] describe correctness pitfalls in MCAS implementations. In this chapter, we carefully prove the correctness of our construction. We also evaluate our construction empirically by comparing to a state-of-the-art MCAS implementation and showing superior performance through a variety of benchmarks (including a production quality B+-Tree [17]) in most considered scenarios.

We note that the delayed unlocking/cleanup introduces a trade-off between higher MCAS performance (due to fewer CASes per MCAS, which also leads to less slow-down due to less helping) and lower read performance (because of the extra level of indirection reads have to traverse when encountering a descriptor left in place after a completed MCAS). One may argue that it also increases the amount of memory consumed by the MCAS algorithm. Regarding

the former, our evaluation shows that the benefits of the lower complexity overcome the drawbacks of indirection in all workloads that experience MCAS contention. Furthermore, to mitigate the impact of indirection in reads, we propose a simple optimization that we describe in Section 5.6.2. As for the latter, we note that much like any lock-free algorithm, the memory consumption of our construction can be tuned by performing memory reclamation more (or less) often.

The rest of the chapter is organized as follows. In Section 5.2 we describe our model. In Section 5.3 we present our impossibility result. Section 5.4 details our MCAS algorithm for volatile memory and Section 5.5 presents the persistent version of our algorithm. Section 5.6 elaborates our lazy memory reclamation scheme for volatile and persistent memory, as well as an optimization to improve read performance. Section 5.7 presents the results of our experimental evaluation. We end with related work in Section 5.8. To improve readability, some content has been moved to the appendix: Appendix D.1 provides the ABA example for the naive simplification of the Harris et al. algorithm and Appendix D.2 contains additional performance graphs.

5.2 System Model

5.2.1 Volatile Memory

We assume a standard model of asynchronous shared memory [116], with basic atomic *read*, *write* and *compare-and-swap* (CAS) operations. The latter receives three arguments—an address, an expected value and a new value; it reads the value stored in the given address and if it is equal to the expected value, atomically stores the new value in the given address, returning the indication of success or failure.

Using those atomic operations, we implement an atomic MCAS operation with the following semantics. The MCAS operation receives an array of tuples, where each tuple contains an address, an expected value and a new value. For ease of presentation, we assume the size of the array is a known constant N. (In practice, the size of the array can be dynamic, and different for every MCAS operation.) The MCAS operation reads values stored in the given addresses, and if they all are equal to respective expected values, atomically writes new values to the corresponding address and returns an indication of success. Otherwise, if at least one read value is different from an expected one, the MCAS operation returns an indication of failure. We also provide a custom implementation of a read operation from a memory location that can be a target of an MCAS operation (which, in the most general case, can be any shared memory location).

Our MCAS implementation is *linearizable* [116]. This means, informally, that each (read or MCAS) operation appears to take effect instantaneously at some point in time in the interval during which the operation executes. In terms of progress, our MCAS implementation is *non-blocking*. That is, a lack of progress of any thread (e.g., due to the suspension or failure

of that thread) does not prevent other threads from applying their operations. Furthermore, the MCAS implementation guarantees *lock-freedom*. That is, given a set of threads applying operations, it guarantees that, eventually, at least one of those threads will complete its operation.

Similar to many non-blocking algorithms, our design makes use of operation descriptors, which store information on existing MCAS operations, including the status of the operation and the array of tuples with addresses and values. We assume each word in the shared memory can contain either a regular value or a pointer to such a descriptor. A similar assumption has been made in prior work on MCAS [88, 106, 219, 226]. In practice, a single (e.g., least significant) bit can be used to distinguish between the two.

Initialization of the descriptor is done before invocation of the MCAS operation. We assume that all the addresses in the descriptor are sorted in a monotonic total order. This assumption is crucial for the liveness property of our algorithm. We note that this assumption can be easily lifted by explicitly sorting the array of tuples by corresponding addresses before an MCAS operation is executed.

5.2.2 Persistent Memory

We extend the model in Section 5.2.1 with standard assumptions about PM [65, 72, 91, 127]. Our PM model is similar to the one used in Chapter 4; we recall it here for completeness. We assume the system is equipped with persistent shared memory that can be accessed through the same set of atomic primitives (read, write and CAS). The system may also be equipped with DRAM to be used as transient storage. As in previous work [127], we assume that the overall system can crash at any time and possibly recover later. On such a full-system crash, we assume that the contents of persistent memory—but not those of processor caches, registers or volatile memory—are preserved. Moreover, threads that are active at the time of the crash are assumed to be lost forever and replaced by new threads in case of recovery. After a full-system crash but before the system recovers and resumes normal execution, we assume a *recovery* routine may be executed, in order to bring persistent memory-resident objects to a consistent state. The recovery routine can be executed in a single thread, and thus it does not have to be thread-safe. Another full-system crash, however, may occur during the recovery routine.

As is standard practice [65, 72, 226], we assume that a priori there is no guarantee on when and in what order cache lines are written back to persistent memory. We assume the existence of two primitives to enforce such write backs. The first primitive is `PERSISTENT_FLUSH(addr)`, which takes as argument a memory location and asynchronously writes the contents of that location to persistent memory. Multiple invocations of this primitive are not ordered with respect to each other and thus several flushes can proceed in parallel. Concrete examples of this primitive are `clflushopt` and `clwb` [123]. The second primitive is `PERSISTENT_FENCE()`, which stalls the CPU until any pending flushes are committed to persistent memory. A concrete

example of this primitive is `sfence` [123]. LOCK-prefixed instructions such as CAS also act as persistent fences [123]. Since persistent flushes do not stall the CPU, whereas persistent fences do, the cost of writing to persistent memory is dominated by the latter instructions and we consider the cost of the former to be negligible.

Regarding initialization, we assume descriptor contents are made persistent before invocation of MCAS.

The safety criterion we use when working with persistent memory is durable linearizability [127]. Informally, an implementation of an object is durably linearizable if it is linearizable and has the following additional properties in case of a full-system crash and recovery: (1) all operations that completed before the crash are reflected in the post-recovery state and (2) if some operation op that was ongoing at the time of the crash is reflected in the post-recovery state, then so are all the operations on which op depends (i.e., operations whose effects op observed and thus need to be linearized before op).

5.3 Impossibility

In this section we show that any lock-free disjoint-access-parallel (DAP) implementation of MCAS requires at least one CAS per modified word. Consider a call to k-CAS($addr_1, \ldots, addr_k$, [old and new values]). We call $addr_1, \ldots, addr_k$ the *set of targets* of the call. We also define the *range* of the call in an execution E to be the set of locations on which CAS (single-word CAS) is performed, successfully or not, during the call in E. Intuitively, we say that an MCAS implementation is *DAP* if non-conflicting calls to k-CAS do not access the same memory locations; for the formal definition, see [124].

Definition 5.3.1 (Star Configuration)**.** *We say that a set* $\{c_0, \ldots, c_\ell\}$ *of calls to* k-CAS *are in a* star configuration *if (1) the sets of targets of* c_0 *and* c_i *are non-disjoint for all* $i \in \{1, \ldots, \ell\}$*, and (2) the sets of targets of* c_i *and* c_j *are disjoint for all* $i \neq j \in \{1, \ldots, \ell\}$.

An example of a star configuration for $\ell = k$ is the following set of calls $\mathscr{C} = \{c_0, \ldots, c_k\}$, where we omit old and new values for ease of notation and we assume that addresses $a_i^{(j)}$ are all distinct:

- c_0: k-CAS($a_1^{(0)}, \ldots, a_k^{(0)}$)
- c_1: k-CAS($a_1^{(0)}, a_2^{(1)}, \ldots, a_k^{(1)}$). Call c_1's set of targets intersects that of c_0 in $a_1^{(0)}$.
- c_i, $1 \leq i \leq k$: k-CAS($a_1^{(i)}, \ldots, a_i^{(0)}, \ldots, a_k^{(i)}$). Call c_i's set of targets intersects that of c_0 in $a_i^{(0)}$ and is disjoint from the set of targets of c_j for all $j \neq i, j \neq 0$.

In this section, we assume without loss of generality that all calls in $\mathscr{C}$ have the correct old values for their target addresses and that each new value is distinct from its respective old

value. Under these assumptions, in every execution it must be that either c_0 succeeds and all $c_1, \ldots, c_k$ fail, or that c_0 fails and all $c_1, \ldots, c_k$ succeed.

We say that a state S of an implementation $\mathscr{A}$ is c_0-valent with respect to (*wrt*) some subset $C \subseteq \mathscr{C}$ if, for any call $c_i \in C$, in any execution starting from S in which only c_0 and c_i take steps, c_0 succeeds. Similarly, we say that a state S is C-valent *wrt* c_0 if, for any call $c_i \in C$, in any execution starting from S in which only c_0 and c_i take steps, c_0 fails. We say that a state is univalent *wrt* c_0 and C if it is c_0-valent or C-valent; otherwise it is bivalent *wrt* c_0 and C. A state is critical *wrt* c_0 and C when (1) it is bivalent *wrt* c_0 and C and (2) if any process in $\{c_0\} \cup C$ takes a step, the state becomes univalent *wrt* c_0 and C.

Note that the initial state of $\mathscr{A}$ must be bivalent *wrt* c_0 and any non-empty subset of $\mathscr{C}$.

Lemma 5.3.2. *Consider a lock-free implementation $\mathscr{A}$ of k-*CAS *and let $\mathscr{C} = \{c_0, \ldots, c_\ell\}$ be a star configuration of calls to k-*CAS*. Then there exists an execution E of $\mathscr{A}$ such that, for all $i \geq 1$, the ranges of c_0 and c_i in E are non-disjoint.*

Proof. We follow a bivalency proof structure. We construct an execution in which process p_i performs call c_i, $i \geq 0$. For ease of notation, we say that "call c_i takes a step" to mean "process p_i takes a step in its execution of c_i".

The execution proceeds in stages. In the first stage, as long as some call in $\mathscr{C}$ can take a step without making the state univalent *wrt* c_0 and any non-empty subset of $\mathscr{C}$, let that call take a step. If the execution runs forever, the implementation is not lock-free. Otherwise, the execution enters a state S where no such step is possible, which must be a critical state *wrt* c_0 and some subset $C_1 \subseteq \mathscr{C} \setminus \{c_0\}$. We choose C_1 to be maximal, i.e., state S is not critical *wrt* c_0 and any subset of $\mathscr{C} \setminus C_1$ (otherwise, add that subset to C_1).

We prove in Lemma 5.3.3 below that c_0 and all calls in C_1 are about to perform CAS on some common location l_1. We let c_0 perform that CAS step, bringing the protocol to state S'. By our choice of C_1 as maximal, S' must be bivalent *wrt* c_0 and any subset of $\mathscr{C} \setminus C_1$. The execution now enters the second stage, in which we let calls in $\mathscr{C} \setminus C_1$ take steps until they reach a critical state *wrt* c_0 and some subset $C_2 \subseteq \mathscr{C} \setminus C_1$. By induction, we can show that eventually c_0 will have reached critical points *wrt* all calls in $\mathscr{C}$. At the end of the execution, we resume each process in $\mathscr{C} \setminus c_0$ for one step; they were each about to perform a CAS step on some location on which c_0 has already performed a CAS step. Thus, in this execution, all calls in $\mathscr{C} \setminus c_0$ have performed a CAS on a common location with c_0. □

Lemma 5.3.3. *Consider a lock-free implementation $\mathscr{A}$ of k-*CAS *and let $\mathscr{C} = \{c_0, \ldots, c_k\}$ be a star configuration of calls to k-*CAS*. If S is a critical state of $\mathscr{A}$ wrt c_0 and some subset $C \subseteq \mathscr{C}$, then in S, c_0 and all calls in C are about to perform a* CAS *step on a common location l.*

Proof. From S, we consider the next steps of c_0 and any $c_i \in C$:

Case 1 One of the calls is about to read; assume *wlog* it is c_0. Consider two possible scenarios. First scenario: c_i moves first and runs solo until it returns (c_i must succeed because c_i took the first step). Second scenario: c_0 moves first and reads, then c_i runs solo until it returns (c_i must fail because c_0 took the first step). But the two scenarios are indistinguishable to c_i, thus c_i must either succeed in both or fail in both, a contradiction.

Case 2 Both calls are about to write. In this case, they must be about to write to the same register r, otherwise their writes commute. First scenario: c_0 writes r, then c_i writes r, then c_i runs solo until it returns (c_i must fail since c_0 took the first step). Second scenario: c_i writes r and then runs solo until it returns (c_i must succeed since c_i took the first step). But the two scenarios are indistinguishable to c_i, since its write to r obliterated any potential write by c_0 to r, so c_i must either succeed in both scenarios or fail in both; a contradiction.

Case 3 c_0 is about to CAS and c_i is about to write (or vice-versa). In this case, their operations must be to the same memory location r (otherwise they commute). First scenario: c_0 CASes r, then c_i writes to r and then runs solo until c_i returns (c_i must fail since c_0 took the first step). Second scenario: c_i writes to r and then runs solo until it returns (c_i must succeed since c_i took the first step). But the two scenarios are indistinguishable to c_i, since its write to r obliterated any preceding CAS by c_0 to r; thus c_i must either succeed in both scenarios or fail in both; a contradiction.

Case 4 Both calls are about to CAS. In this case, they must be about to CAS the same location, otherwise their CASes commute. □

Theorem 5.3.4. *Consider a lock-free disjoint-access-parallel implementation $\mathscr{A}$ of k-*CAS *in a system with $n > k$ processes. Then there exists some execution E of $\mathscr{A}$ such that in E some call to k-*CAS *performs* CAS *on at least k locations.*

Proof. We prove the theorem by contradiction. We first assume that calls to k-CAS perform CAS on *exactly* $k-1$ locations and derive a contradiction; we later show how assuming that k-CAS performs CAS on *at most* $k-1$ locations also leads to a contradiction.

We construct an execution E in which two concurrent but non-contending k-CAS calls (i.e., two k-CAS calls with disjoint sets of targets) perform CAS on the same location, thus contradicting the disjoint-access-parallelism (DAP) property and proving the theorem.

Let $c_0, \ldots, c_k$ be $k+1$ calls to k-CAS in a star configuration. By Lemma 5.3.2, there exists an execution E of $\mathscr{A}$ such that, for all $i \geq 1$, the ranges of c_0 and c_i in E are non-disjoint.

Let $l_1, \ldots, l_{k-1}$ be the range of c_0. By Lemma 5.3.2, in E the range of c_1 must intersect that of c_0 in at least one location; assume *wlog* it is l_1. Furthermore, the range of c_2 must also intersect that of c_0 in at least one location; moreover, due to the DAP property, the intersection must contain some location other than l_1, since c_1 and c_2 have disjoint sets of targets. By induction, we can show that the range of each call $c_i, i \in \{1, 2, \ldots, k-1\}$ intersects the range of

c_0 in l_i. However, the range of c_k must also intersect the range of c_0 in some location other than $l_1, \ldots, l_{k-1}$, due to the DAP property. We have reached a contradiction.

If we now assume that calls to k-CAS perform CAS on $k-1$ *or fewer* locations, then we also reach a similar contradiction as above. In fact, if some call c_i performs CAS on strictly fewer than $k-1$ locations, this may cause the contradiction to occur before call c_k, as c_i now has fewer locations to choose from in order intersect with the range of c_0 in some location that is not in the ranges of $c_1, \ldots, c_{i-1}$. □

5.4 Volatile MCAS with $k+1$ CAS

In this section we describe our MCAS construction for volatile memory. Our algorithm uses $k+1$ CAS operations in the common uncontended case, and does not involve cleaning up after completed MCAS operations. In Section 5.6 we describe a memory management scheme that can be used to clean up after completed MCAS operations as well as for reclaiming or reusing operation descriptors employed by the algorithm.

5.4.1 High-level Description

As is standard practice [102, 106, 219], our MCAS construction supports two operations: MCAS and `read`. Similarly to most MCAS algorithms [102, 106, 219], the MCAS operation uses operation descriptors that contain a set of addresses (the *target* addresses or words), and *old* and *new* values for each target address. In addition, each operation descriptor contains a *status* word indicating the status of the corresponding MCAS operation.

The MCAS operation proceeds in two stages. In the first stage, we attempt to install a pointer to the operator descriptor in each memory word targeted by the MCAS operation. If we succeed to install the pointer, we say that the target address is *owned* (or *locked*) by the descriptor. The first stage ends when all target addresses are owned by the descriptor, or if we find a target address with a value different from the expected one. In the second stage, we *finalize* the MCAS operation by atomically changing its status to indicate its success or failure, depending on whether the first stage was successful (i.e., all target addresses have been locked). The `read` operation returns the current value at an address, either by reading it directly from the target address or by reading the appropriate value from a descriptor of a completed MCAS operation installed in that address. If either MCAS or `read` encounter another MCAS in progress (e.g., when they attempt to read the current value in the target address), they first help that MCAS operation to complete.

5.4.2 Technical Details

Structures and Terminology. We describe the structures used by our algorithm and explain the terminology. Pseudocode for the structures is shown in Listing 5.1. An `MCASDescriptor`

Listing 5.1 – Data structures used by our MCAS algorithm for volatile memory.

```
struct WordDescriptor {
    void* address;
    uintptr_t old;
    uintptr_t new;
    MCASDescriptor* parent;
};

enum StatusType { ACTIVE, SUCCESSFUL, FAILED };

struct MCASDescriptor {
    StatusType status;
    size_t N;
    WordDescriptor words[N];
};
```

Algorithm 5.2 – The `readInternal` auxiliary function, used by our MCAS algorithm for volatile memory.

```
1  readInternal(void* addr, MCASDescriptor *self) {
2  retry_read:
3      val = *addr;
4      if (!isDescriptor(val)) then return <val,val>;
5      else { // found a descriptor
6          MCASDescriptor* parent = val->parent;
7          if (parent != self && parent->status == ACTIVE) {
8              MCAS(parent);
9              goto retry_read;
10         } else {
11             return parent->status == SUCCESSFUL ?
12                 <val,val->new> : <val,val->old>;
13 } } }
```

describes an MCAS operation. It contains a status field, which can be `ACTIVE`, `SUCCESSFUL` or `FAILED`, the number `N` of words targeted by the MCAS and an array of `WordDescriptors` for those words. These `WordDescriptors` are the *children* of the `MCASDescriptor`, who is their *parent*. We say that an `MCASDescriptor` (and the MCAS it describes) is *active* if its status is `ACTIVE` and *finalized* otherwise.

The `WordDescriptor` contains information related to a given word as target of an MCAS operation: the word's address in memory, its expected value and the new intended value. The `WordDescriptor` also contains a pointer to the descriptor of its parent MCAS operation. As described later, the pointer is used as an optimization for fast lookup of the status field in the `MCASDescriptor`, and can be eliminated.

Algorithm. Both MCAS and `read` operations rely on the auxiliary `readInternal` function shown in Algorithm 5.2. The `readInternal` function takes an address `addr` and an

Algorithm 5.3 – Our MCAS algorithm for volatile memory. Commands in *italic* are related to memory reclamation (discussed in a later section).

```
14 read(void* address) {
15     epochStart();
16     <content, value> = readInternal(address, NULL);
17     epochEnd();
18     return value; }

20 MCAS(MCASDescriptor* desc) {
21     epochStart();
22     success = true;
23     for wordDesc in desc->words {
24 retry_word:
25         <content, value> = readInternal(wordDesc.address, desc);
26         // if this word already points to the right place, move on
27         if (content == &wordDesc) continue;
28         // if the expected value is different, the MCAS fails
29         if (value != wordDesc.old) { success = false; break;  }
30         if (desc->status != ACTIVE) break;
31         // try to install the pointer to my descriptor; if failed, retry
32         if (!CAS(wordDesc.address, content, &wordDesc)) goto retry_word;
33     }
34     if (CAS(&desc.status, ACTIVE, success ? SUCCESSFUL : FAILED)) {
35         // if I finalized this descriptor, mark it for reclamation
36         retireForCleanup(desc); }
37     returnValue = (desc.status == SUCCESSFUL);
38     epochEnd();
39     return returnValue; }
```

`MCASDescriptor self` (called the *current descriptor*) and returns a tuple. The tuple contains two values (which might be identical), and, intuitively, represent the contents in the given (target) address and the actual value the former represents. More specifically, `readInternal` reads the content of the given `addr` (Line 3). If `addr` does not point to a descriptor (this is determined by the `isDescriptor` function; see below), the returned tuple contains two copies of the contents of `addr` (Line 4). If `addr` points to an active `WordDescriptor` whose parent is not the same as `self`, then `readInternal` helps the other (MCAS) operation to complete (Line 8) and then restarts (Line 9). Therefore, the role of the `self` pointer is to avoid an (MCAS) operation to help itself recursively. If `addr` points to a finalized descriptor, the tuple returned by `readInternal` contains the pointer to the descriptor and the final value, corresponding to the status of the descriptor (Line 12). Finally, if `addr` points to a descriptor whose parent is equal to `self`, then `readInternal` returns the pointer to that descriptor (Line 12; a value is also returned in the tuple in this case, but is disregarded; see below).

Algorithm 5.3 provides the pseudo-code for the `read` and MCAS operations. The pseudo-code includes extensions relevant to memory management (in *italics*), whose discussion is deferred to Section 5.6.

The `read` operation is simply a call to `readInternal` with a `self` equal to `null` as the current

operation descriptor (Line 16).

The MCAS operation takes as argument an `MCASDescriptor` and returns a boolean indicating success or failure. As mentioned above, the operation proceeds in two stages. In the first stage, MCAS attempts to take ownership of (or *acquire*) each target word (Lines 23–33). To this end, for each `WordDescriptor` w in its `words` array, we start by calling `readInternal` on w's target address `addr` (Line 25; as described above, this handles any helping required in case another active operation owns `addr`). If `addr` is already owned by the current MCAS, we move on to the next word (Line 27). Otherwise, if the current value at `addr` does not match the expected value of w, the MCAS cannot succeed and thus we can skip the next `WordDescriptors` and go to the second stage (Line 29). If the values do match, we re-check if the operation is still active (line 30); otherwise we go to the second stage—this prevents a memory location from being re-acquired by the current operation op in case op was already finalized by a helping thread. Finally, we attempt to take ownership of `addr` through a CAS (Line 32). Note that the failure of this CAS might mean that another thread has concurrently helped this MCAS to lock the target word. Therefore, we simply retry taking ownership on this target word, rather than failing the MCAS operation (Line 32).

In the second stage (Lines 34–36), MCAS finalizes the descriptor by atomically changing its status from `ACTIVE` to `SUCCESSFUL` (if all word acquisitions were successful in stage one) or to `FAILED` (otherwise).

Our pseudocode assumes the existence of the `isDescriptor` function, which takes a value and returns `true` if and only if the value is a pointer to a `WordDescriptor`. This function can be implemented, for instance, by designating a low-order *mark bit* in a word to indicate whether it contains a pointer to a descriptor or not [106, 226]. Whenever we make an address point to a descriptor (e.g., Line 32) or convert the contents of a word into a pointer to descriptor (e.g., Line 6), we also set or unset the mark bit, respectively. In the interest of clarity, we do not show the implementation of `isDescriptor` or the code for marking/unmarking pointers.

5.4.3 Correctness

In this section we argue that our MCAS algorithm is linearizable and lock-free. We give preliminary invariants before showing the main results.

Lemma 5.4.1. *Once an* MCAS *descriptor is finalized, its status never changes again.* The status can only be modified through the CAS at Line 34, whose expected value is `ACTIVE`. If the CAS succeeds, the new value of the status can only be `SUCCESSFUL` or `FAILED`, thus any subsequent attempt to change the status will fail.

Lemma 5.4.2. *An* MCAS *descriptor is finalized by at most one thread.* This follows from the fact that a descriptor is finalized through a CAS and the fact that an MCAS descriptor cannot change status after being finalized (Lemma 5.4.1).

Lemma 5.4.3. *If at least one thread attempts to finalize a descriptor d, some thread will success-*

fully finalize d. The initial status of a descriptor is `ACTIVE`. Any thread attempting to finalize a descriptor does so through the CAS at Line 34, with expected value `ACTIVE`. Thus, at least one CAS finds the status to be `ACTIVE` and successfully changes it.

Lemma 5.4.4. *An* MCAS *descriptor d is finalized as successful only if some thread observed all target locations of d to be acquired by d.* This is because the status is changed to successful only if the `success` variable is true at Line 34. This only happens if some thread completed the for-loop over all of d's `WordDescriptors` without exiting the loop at Line 29. The only two ways for a thread to move to the next `WordDescriptor` in the loop is if the thread sees the current target location was already acquired by d (Line 27) or if the thread successfully acquired the current target location for d (Line 32). In both cases the thread observed the target location to be acquired by d.

Lemma 5.4.5. *An* MCAS *descriptor d is finalized as failed only if some thread observed a target location of d to contain a different value than its expected value in d.* This is because the only way for the status to be changed to failed is if the `success` variable is false. This only happens if some thread observed the current value of a target location is different from its expected value in Line 29.

Lemma 5.4.6. *After a location l becomes acquired by some operation op, l will never become un-acquired again.* This is because the only instruction that modifies a location l is the acquire CAS at Line 32.

We say that an operation op_1 *helps* an MCAS operation op_2 if op_1 calls MCAS with op_2's descriptor in Line 8.

Lemma 5.4.7. *After a location l becomes acquired by some operation op, no operation $op' \neq op$ will acquire l before op becomes finalized.* Assume by contradiction that op' acquires l after op acquires l and while op is still active. Consider op' last call to `readInternal` (Line 25) before the successful acquisition of l. During this call, op' must have observed that l is owned by op (otherwise; if op had acquired l after the call to `readInternal`, the acquisition CAS would have failed). Moreover, op was active during that call to `readInternal` by op'. Thus, op' helped op before returning from `readInternal`, finalizing op in the process. Thus op cannot be active at the time of the acquisition, a contradiction.

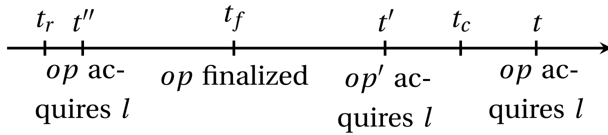
We say that a location l is *re-acquired* by operation op at time t if (1) l becomes acquired by op at time t, (2) there exists time $t' < t$ such that l became acquired by $op' \neq op$ at time t', and (3) there exists time $t'' < t'$ such that l became acquired by op at time t''.

Lemma 5.4.8. *A location cannot be re-acquired.* Assume the contrary and let t be the earliest time when any location is re-acquired in a given execution E. Let l be that location and op be the operation re-acquiring it. This means that l became acquired by op at some time t'', then became acquired by some $op' \neq op$ at time $t' > t$ and then later became acquired

by op at time $t > t'$ (the times in this lemma are represented in the figure below, for convenience). By Lemma 5.4.7, op must have become finalized at some time $t_f < t'$ (t_f is unique by Lemma 5.4.1).

Now consider the thread T which acquires l on behalf of op at time t. T does so through the CAS at line 32. Since op becomes finalized at time t_f, T must have performed the status check at line 30 at some time $t_s < t_f$ (otherwise T would have exited from the for loop without acquiring l). Let $t_r < t_s$ be the last time when T performed `readInternal` (line 25) before t_s. Note that $t_r < t''$, otherwise T would have seen l as already acquired by op at line 27 and continued without attempting to acquire l.

Let $\langle c, v\rangle$ be the return value of the `readInternal` call by T at t_r; this means that l's value was c at some time before t_r. Since T successfully performs the CAS at line 32 at time t, the value of l must also be c immediately before t. However, the acquisition of l at time $t'' > t_r$ changes the value of l from c. Therefore, it must be the case that some thread changes the value of l back to c at some time t_c between t'' and t. Note that c must be a word descriptor (due to Lemma 5.4.6). Since word descriptors are unique, they uniquely identify their parent operations. Therefore, l must have been owned by some operation op'' before t_r and again at t_c; this means that op'' *re-acquired* l at time t_c, contradicting our choice of t as the earliest re-acquisition time.



Lemma 5.4.9. *A location l cannot be acquired by operation op after op is finalized as successful.* This follows from Lemmas 5.4.4 and 5.4.8.

Lemma 5.4.10. *If op_1 helps op_2, then either op_2 highest acquired location is higher than op_1's highest acquired location, or op_1 has not acquired any locations.* If op_1 is a read operation, the statement is trivially true. Assume now that op_1 is an MCAS operation that helps op_2 and that op_1's highest acquired location is higher than op_2's highest acquired location ($\star$). Since op_1 helps op_2, op_1 has observed one of its target locations l to be already acquired by op_2. But since op_1 iterates over locations in increasing order, l must be higher than op_1's highest acquired location. This contradicts $\star$.

We define the *helping graph at time t*, $H(t)$, as follows. The vertices of $H(t)$ are the ongoing operations at time t. There is an edge from op_1 to op_2 if op_1 is helping op_2 at t. We define the *call depth* of an operation op at time t to be the length of the longest path starting from op in $H(t)$.

Lemma 5.4.11. *For any operation op and any time t, the call depth of op at t is finite.* Assume the contrary. Since each thread can have at most one ongoing operation at t, $H(t)$ has a finite

vertex set. Let op be an operation and t be a time such that op has an infinite call depth at t. Then, $H(t)$ must contain a cycle. This is a contradiction: if the cycle contains an operation op_0 that has no acquired locations, then op_0's predecessor in the cycle cannot be helping it; if the cycle does not contain such an operation, then by traversing the cycle we would find operations with strictly increasing highest acquired locations (Lemma 5.4.10).

Informally, Lemma 5.4.11 says that while in our algorithm it is possible for operations to recursively help one another, the recursion depth is finite at any time, due to the sorting of memory locations.

We define the following predicates (recall that n is the number of threads). Let $S(k)$: "If there are $0 < k \leq n$ concurrent operations and at least one thread is taking steps and no operations are created, at least one operation will eventually return". Let $P(k)$: "If there are $0 < k \leq n$ concurrent operations and at least one thread is taking steps, at least one operation will eventually return".

Lemma 5.4.12. *$S(k)$ is true for all k, $0 < k \leq n$.* Assume the contrary. Pick an active thread T: T is taking steps infinitely often, but no operations ever return. By Lemma 5.4.11, the call depth of T is finite, thus T must be taking some backward branch infinitely often. If T is taking the branch at Line 9 infinitely often, then MCAS operations are being finalized infinitely often (Line 9 is only executed if some operation was active at Line 7; but that same operation must be finalized by Line 9 due to the preceding MCAS call which returns only after the operation is finalized). This is a contradiction because we started with a finite number of MCAS operations and no operations are being created. If T is taking the branch at Line 32 infinitely often, then locations either (a) become acquired infinitely often or (b) change owners infinitely often. Both possibilities lead to a contradiction: (a) because there are a finite number of target locations of ongoing MCAS operation and locations never become unacquired and (b) because locations change owners only after operations become finalized, which would imply that operations become finalized infinitely often.

Lemma 5.4.13. *$P(k)$ is true for all k, $0 < k \leq n$.* Consider the case $k = n$. $P(k)$ is equivalent to $S(k)$ in this case (no operations can be created if there are already as many operations as threads), and thus true. Consider the case $k = n-1$. If some operation is eventually created, then eventually some operation will return, by $P(n)$. If no operation is ever created, then eventually some operation will return, by $S(n-1)$. We can continue in this manner with $k = n-2, \ldots, 1$, each time using either $P(k+1)$ or $S(k)$.

Lemma 5.4.14. *Our implementation is lock-free.* This follows immediately from Lemma 5.4.13.

Lemma 5.4.15. *Linearization point of a failed* MCAS. By Lemma 5.4.5, if descriptor d is finalized as failed by thread T at time t_1, then at time $t_0 < t_1$, T has observed some target location l to contain a different value than l's expected value in d. We can take t_0 as the linearization point of the MCAS.

Lemma 5.4.16. *Linearization point of a successful* MCAS. By Lemma 5.4.4, if thread T changes the status of descriptor d to successful, then T previously observed all of d's target locations

to be acquired by d. Thus, when changing the status of d to successful, T changes the logical values of all target locations, marking the linearization point.

Lemma 5.4.17. *Linearization point of a read.* The linearization point of a read is the last executed dereference instruction at Line 3.

5.5 Persistent MCAS with $k+1$ CAS and 2 Persistent Fences

We discuss the modifications required to make our volatile MCAS algorithm work with persistent memory. The extra instructions are shown underlined in Algorithms 5.4 and 5.5.

Algorithm 5.4 – The `readInternal` auxiliary function, used by our MCAS algorithm for persistent memory. Underlined commands are related to persistence.

```
1  readInternal(void* addr, MCASDescriptor *self) {
2  retry_read:
3      val = *addr;
4      if (!isDescriptor(val)) then return <val,val>;
5      else { // found a descriptor
6          MCASDescriptor* parent = val->parent;
7          if (parent != self) && parent->status == ACTIVE) {
8              MCAS(parent);
9              goto retry_read;
10         } else if (parent->status & DirtyFlag) {
11             PERSISTENT_FLUSH(&parent->status);
12             PERSISTENT_FENCE();
13             parent->status = parent->status & ~DirtyFlag;
14             goto retry_read;
15         } else {
16             return parent->status == SUCCESSFUL ?
17                 <val,val->new> : <val,val->old>;
18 }  }  }
```

In the `MCAS` function (Algorithm 5.5), after all target locations have been successfully acquired, we add one persistent flush per target word and one persistent fence overall. The persistent fence ensures that all target locations persistently point to their respective `WordDescriptors` before attempting to modify the status.

When finalizing the status in line 41, we mark the status with a special `DirtyFlag`. This flag indicates that the status is not yet persistent. We then perform a persistent flush and fence after the status has been finalized. This ensures that the finalized status of the descriptor is persistent before returning from the MCAS. Finally, we unset the `DirtyFlag` with a simple store (line 46); this store cannot create a race with the CAS in line 41 because that CAS must fail (the status must be already finalized if some thread is already at line 46).

We also modify the `readInternal` function (Algorithm 5.4) such that, when an operation op encounters another operation op' whose status is finalized but still has the `DirtyFlag` set, op helps op' persist its status and unsets the `DirtyFlag` on op' status.

Algorithm 5.5 – Our MCAS algorithm for persistent memory. Commands in *italic* are related to memory reclamation, and underlined commands are related to persistence.

```
19  read(void* address) {
20      epochStart();
21      <content, value> = readInternal(address, NULL);
22      epochEnd();
23      return value; }

25  MCAS(MCASDescriptor* desc) {
26      epochStart();
27      success = true;
28      for wordDesc in desc->words {
29  retry_word:
30          <content, value> = readInternal(wordDesc.address, desc);
31          // if this word already points to the right place, move on
32          if (content == &wordDesc) continue;
33          // if the expected value is different, the MCAS fails
34          if (value != wordDesc.old) { success = false; break;  }
35          if (desc->status != ACTIVE) break;
36          // try to install the pointer to my descriptor; if failed,
              ↪ retry
37          if (!CAS(wordDesc.address, content, &wordDesc)) goto retry_word;
              ↪ }
38      for wordDesc in desc->words { PERSISTENT_FLUSH(wordDesc.address); }
39      PERSISTENT_FENCE();
40      newStatus = success ? SUCCESSFUL : FAILED;
41      if (CAS(&desc.status, ACTIVE, newStatus | DirtyFlag)){
42          // if I finalized this descriptor, mark it for reclamation
43          retireForCleanup(desc); }
44      PERSISTENT_FLUSH(&desc.status);
45      PERSISTENT_FENCE();
46      parent->status = parent->status & ~DirtyFlag;
47      returnValue = (desc.status == SUCCESSFUL);
48      epochEnd();
49      return returnValue; }
```

Our modifications enforce the following invariants. First, at the time when a descriptor becomes finalized, its acquisitions of target locations are persistent. Second, at the time when an MCAS operation returns, its finalized status is persistent. Third, when a read or MCAS operation op returns, all operations on which op depends are finalized and their statuses are persistent. With these invariants, we can argue that our persistent MCAS is correct. By correctness we refer to lock-freedom (liveness) and durable linearizability (safety). Lock-freedom is clearly preserved by our additions, thus we focus on durable linearizability. We examine the point in time when a full-system crash may occur during the execution of an MCAS operation op. There are two possibilities to consider:

1. If the crash occurs before op's status was finalized and made persistent, then we know that no operation op' which observed the effects of op could have returned before the crash; otherwise, op' would have helped op and persisted its status. In this case, neither

op nor any such op' will be linearized before the crash; during recovery, their effects will be rolled back by reverting any acquired locations to their old values.

2. If the crash occurs after op's status was finalized and made persistent, then op is linearized before the crash. During recovery, any locations still acquired by op will be detached and given either their new or old values (depending on op's success or failure status), as specified in op's descriptor.

In sum, the recovery procedure of our algorithm is as follows. The recovery goes through each operation descriptor D. If D's status is not finalized, then we roll D back by going through each target location ℓ of D; if ℓ is acquired by D (i.e., points to D), then we write into ℓ its old value, as specified in D. If D's status is finalized, then we detach D and install final values; we go through each target location ℓ of D; if ℓ is acquired by D and D was successful (resp. failed), then we write into ℓ the new (resp. old) value as specified in D.

5.6 Memory Management

The MCAS algorithm has been presented so far under the assumption that no memory is ever reclaimed. For practical considerations, however, one should to be able to reclaim and/or reuse MCAS operation descriptors. While efficient memory management of concurrent data structures remains an active area of research (see, e.g., [9, 43, 78, 200, 228]), here we describe one possible mechanism suitable for an MCAS implementation. We first describe memory management in the context of the volatile MCAS implementation presented in Section 5.4; we then explain how to extend our scheme to the persistent implementation in Section 5.5, as well as a possible optimization to make reads more efficient.

We note that the life cycle of an operation descriptor comprises several phases. Once its status is no longer `ACTIVE`, the (finalized) descriptor cannot be recycled just yet as certain memory locations can point to it. Therefore, we need first to *detach* such a descriptor by replacing the pointers to the descriptor (using CAS) with actual values (respective to whether the corresponding MCAS has succeeded or failed) in affected memory locations. Only after that, a detached descriptor can be recycled, provided no concurrently running thread holds a reference to it. Note that CASes in the detachment phase are necessary only for those affected memory locations that still point to the to-be-detached descriptor, which, as our evaluation shows, is rare in practice.

Our scheme keeps track of two categories of descriptors: (1) those that have been finalized but not yet detached and (2) those that have been detached but to which other threads might still hold references. Similar to RCU approaches [171, 173], we use thread-local epoch counters to track threads' progress and infer when a descriptor can be moved from category (1) to category (2), and when a descriptor from category (2) can be reclaimed.

We use two thread-local lists for reclaiming operation descriptors: One list is for descrip-

tors that have been finalized, but not detached yet (finalizedDescList), and another is for descriptors that have been detached but to which readers might still hold references (detachedDescList).

In general, our memory management scheme is similar to an RCU (read-copy-update) implementation [171, 173]. We start with a simple blocking scheme, extending it into a non-blocking one. Each thread maintains an epoch number, incremented by the thread upon the entry to and before the exit from the read and MCAS functions (see, e.g., Lines 15 and 17 in Algorithm 5.3). In retireForCleanup function (cf. Line 36 in Algorithm 5.3), a thread adds the given descriptor to finalizedDescList. Once the size of this list reaches a certain threshold, the thread invokes a function similar semantically to synchronize_rcu() [171]. That is, it runs through all thread epochs, and waits for every epoch with an odd value (indicating that a thread is inside the read or MCAS functions) to advance. Once all epochs are traversed, all descriptors currently in the detachedDescList list can be reclaimed (returned to the operating system or put into a list of available descriptors for reuse). At the same time, all descriptors currently in the finalizedDescList list can be moved to the detachedDescList list, after replacing pointers to those descriptors in corresponding memory locations with their actual values.

To elaborate on this last step, given an MCASDescriptor descriptor d that is about to be moved from finalizedDescList to detachedDescList, a thread runs through all the WordDescriptors stored in d. For every such WordDescriptor w, the thread checks whether w->address is equal to d and if so, writes w->old or w->new into w->address according to the status of d. The check and the write are done atomically using CAS.

The scheme presented so far is blocking—if a thread does not advance its epoch number, any thread will be unable to complete the traversal of epochs. To avoid this issue, each thread may store a local copy of all thread epochs it has seen during the last traversal. On its next epoch traversal, it compares the current and the previously seen epochs for each thread t, and if those two are different, it infers that t has made progress. If progress is detected for all threads, any descriptor that was placed into finalizedDescList (detachedDescList) before the previous epoch traversal can be detached (reclaimed, respectively).

Note that while this scheme is non-blocking, a failure of a thread might prevent reclamation of *any* memory associated with descriptors. This is a common issue with epoch-based reclamation schemes [78], which could be resolved either by enhancing the scheme (e.g., as in [43]) or by switching to a different scheme, e.g., one based on hazard pointers [78, 178].

5.6.1 Managing Persistent Memory

We now show how the memory management mechanism described above is extended to manage persistent memory. Upon recovery from a crash, any pending PMCAS operation is completed as described in Section 5.5. Pending PMCAS operations can be found by scanning

allocated descriptors (e.g., if descriptors are allocated from a pool, similar to David et al. [72]). Moreover, since we assume the recovery is done by a single thread, we can immediately detach and recycle any finalized descriptor (after writing back the actual values into corresponding memory locations). Therefore, when considering persistent memory, the only change required to support correct recycling of descriptors (in addition to using a persistent memory allocator) is flushing all writes while detaching descriptors and introducing a persistent fence right before reclaiming descriptors from `detachedDescList`. The fence is required to avoid a situation where a detached descriptor is recycled and a crash happens while the descriptor is being initialized with new values. In this case, and if a fence is not used, some memory locations may still point to the descriptor (since updates to those locations might have not been persisted before the crash), while the descriptor may already be updated with new content. Note, though, that the flushes and the fence take place off the critical path, therefore their impact on the performance of PMCAS is expected to be negligible.

5.6.2 Efficient Reads

Once a memory location has been modified by an MCAS operation, even if by a failed one, it would refer to an operation descriptor until that descriptor is detached. Until that happens, the latency of a read operation from that memory location would be increased as it would have to access an operation descriptor to determine the value that needs to be returned by the read. The memory management mechanism as described above, however, would detach the descriptor only as a part of an MCAS operation. This might cause degraded performance for read-dominated workloads in which MCAS operations are rare.

To this end, we propose the following optimization for eventual removal of references to an operation descriptor and storing the corresponding value directly in the memory location as part of the read operation. If a read operation finds a pointer to a finalized operation descriptor, it will generate a pseudo-random number[1]. With a small probability, it will run a simplified version of the memory reclamation scheme described above. Specifically, it will scan epochs of all other threads, and then change the contents of the memory location it attempts to read to the actual value (using CAS). (To avoid deadlock between two threads scanning epoch numbers, a thread may indicate that it is in the middle of an epoch scan so that any descriptor can be detached, but not recycled at that time.)

5.7 Evaluation

5.7.1 Experimental Setup

We evaluate our algorithm on a 2-socket Intel Xeon machine with two E5-2630 v4 processors operating at 3.1 GHz. Each processor has 10 cores, each core has 2 hardware threads

[1]Generating a local pseudo-random number is a relatively inexpensive operation that requires only a few processor cycles (see, for instance, the generator in ASCYLIB [18].)

(40 hardware threads total). Each experimental run lasts 5 seconds; shown values are the average of 5 runs. We base our evaluation on the framework available from the authors of PMwCAS [198, 226].

The baseline of our evaluation is the volatile version of PMwCAS [198, 226], a state-of-the-art implementation of the Harris et al. [106] algorithm. Like the Harris et al. algorithm, volatile PMwCAS requires $3k+1$ CASes per k-CAS. We use PMwCAS as our baseline since (1) it has recent, openly available and well-maintained code and (2) it is to our knowledge the only other MCAS algorithm in which readers do not write to shared memory in the common uncontended case.

PMwCAS implements an optimization of the Harris et al. algorithm: it marks pointers with a special *RDCSS* flag instead of allocating a distinct RDCSS descriptor. However, we found that this optimization made the PMwCAS algorithm incorrect, due to an ABA vulnerability. In our evaluation, we fixed the PMwCAS implementation to allocate and manually manage RDCSS descriptors.

Our evaluation uses three benchmarks: an *array benchmark* in which threads perform MCAS-based read-modify-write operations at random locations in an array, a *doubly-linked list benchmark*, in which threads perform MCAS-based operations on a list implementing an ordered set, and a *B+-tree benchmark* in which threads perform MCAS-based operations on a B+-tree. The first two benchmarks are based on the implementation available in [198], and the third is based on PiBench [197] and BzTree [17, 46]. We note, however, that we modified the benchmark in [198] so all threads operate on the same key range (rather than having each thread using a unique set of keys), so we could induce contention by controlling the size of the key range.

In each experiment, we vary the number of threads from 1 to 39 (we reserve one hardware thread for the main thread). Threads are assigned according to the default settings in the evaluation frameworks used [46, 197, 198]. In the array and list benchmarks, threads are assigned in the following way: we first populate the first hardware thread of each core on the first socket, then on the second socket, then we populate the second hardware thread on each core on the first socket, and finally the second hardware thread on each core on the second socket. The B+-tree benchmark uses OpenMP [188], which dictates thread assignment; it also employs a scalable memory allocator [2].

5.7.2 Array Benchmark

The benchmark consists of each thread performing the following in a tight loop: reading k locations at random from the array ($k = 4$ in our experiments), computing a new value for each location, and attempting to install the new values using an MCAS.

In this benchmark we measure two quantities. The first is throughput: the number of read-modify-write operations completed successfully per time unit. The second metric is the

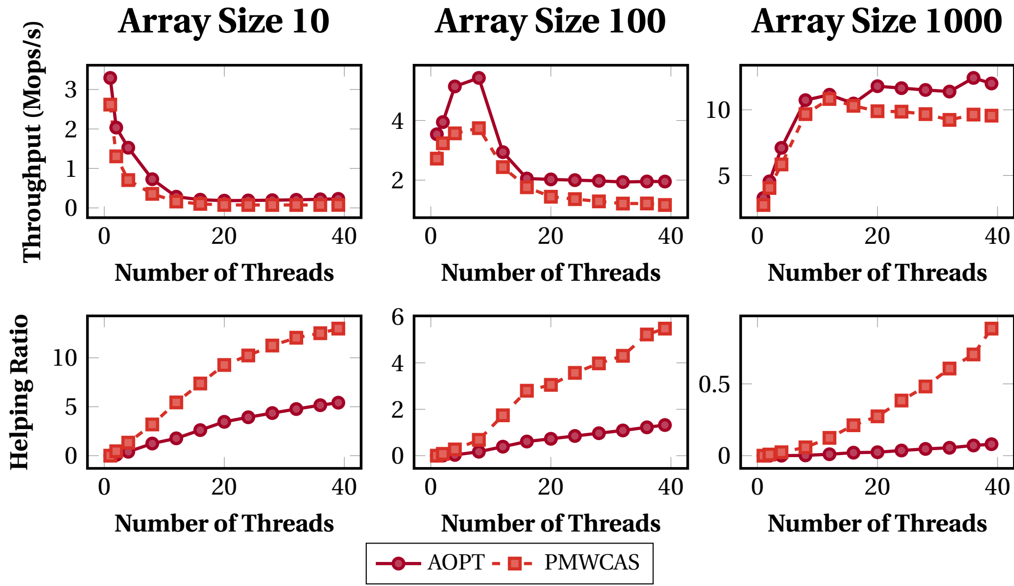


Figure 5.1 – Array benchmark. Top row shows throughput (higher is better), bottom row shows helping ratio (lower is better). Each column corresponds to a different array size (10, 100 and 1000, respectively).

helping ratio. We measure the helping ratio by dividing the number of *ongoing* MCAS operations encountered (and helped) during read or MCAS operations by the total number of MCAS operations. A higher helping ratio thus means more operations are slowed down due to the need to help other, incomplete MCAS operations.

We run the benchmark with three array sizes (10, 100, and 1000) in order to capture different contention levels. The results of this benchmark are shown in Figure 5.1 (our algorithm is denoted *AOPT* in all figures in this section).

The top row of Figure 5.1 shows that our algorithm outperforms PMwCAS at every contention level and at every thread count, including in single-threaded mode. This can be explained by two related factors. First, our algorithm has a lower CAS complexity ($k+1$ CASes per k-CAS for our algorithm compared to $3k+1$ for PMwCAS). Second, as a consequence of its lower complexity, in our algorithm there is a shorter “window” for each MCAS operation to interfere with other operations by forcing them to help.

To illustrate the second factor above, we examine the helping ratios of the two algorithms (bottom row of Figure 5.1). We observe that the helping ratio of our algorithm is considerably lower than that of PMwCAS. This means that, on average, each operation helps (and is slowed down by) fewer MCAS operations in our algorithm than in PMwCAS.

In order to quantify the impact of descriptor cleanup on performance in our algorithm, we also measure the *detaching ratio*: the number of CASes performed in order to detach (in the sense of Section 5.6) finalized MCAS descriptors, divided by the total number of completed

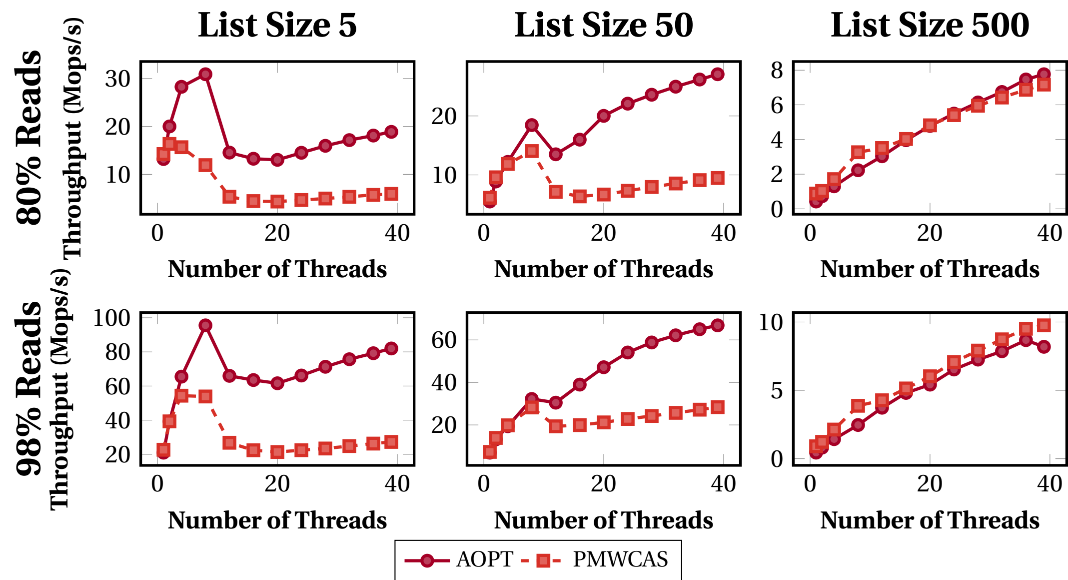


Figure 5.2 – Doubly-linked list benchmark. Top row shows results for 80% reads workload; bottom row shows results for 98% reads workload. Each column corresponds to a different initial list size (5, 50 and 500 elements, respectively).

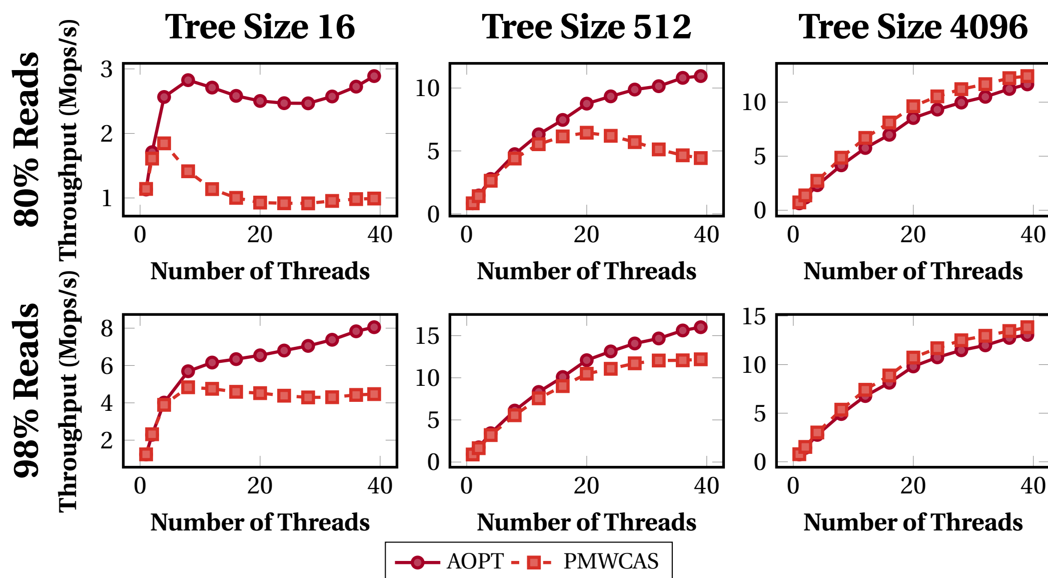


Figure 5.3 – B+-tree benchmark. Top row shows results for 80% reads workload; bottom row shows results for 98% reads workload. Each column corresponds to a different initial tree size (16, 512 and 4096 elements, respectively).

MCAS operations. We find the detaching ratio to be less than 0.001 for every thread count and array size. This is because finalized MCAS descriptors are constantly being replaced by ongoing MCAS operations, and thus recycling these detached descriptors requires no CASes. We conclude that the vast majority of our MCAS operations do not incur any cleanup CASes.

5.7.3 Doubly-linked List Benchmark

In this benchmark we operate on a shared ordered set object implemented from a doubly-linked list. The list supports search and update (insert and delete) operations. Insertions are done using 2-CAS and deletions are done using 3-CAS. We initialize the list by inserting a predefined (configurable) number of nodes. During the benchmark, each thread selects an operation type (search, insert or delete) at random, according to a configurable distribution; the thread also selects a value at random; it then performs the selected operation with the selected value.

We perform this benchmark with three initial list sizes (5, 50 and 500 elements) and two operation distributions: (1) 80% reads, 20% updates (in all our experiments, updates are evenly distributed among insertions and deletions) and (2) 98% reads. As is standard practice, the initial size of the list is half of the key range. Results are shown in Figure 5.2. We also ran experiments with 50% reads and 100% reads; for better readability, performance graphs for these less representative cases are deferred to Appendix D.2.

With 80% reads, our algorithm outperforms PMwCAS for list sizes 5 and 50 by 2.6× and 2.2× on average, respectively. This shows that under high and moderate contention, our algorithm's faster MCAS operations (due to the double effect of lower complexity and lower helping ratio) compensate for its slower read operations (due to the extra level of indirection), even in read-heavy workloads. In the low contention case (list size 500), PMwCAS outperforms our algorithm at low thread counts and is outperformed at high thread counts. On average, PMwCAS outperforms our algorithm by 1.2×. Under low contention, operations have a low probability to conflict on the same element and thus the lower read complexity of PMwCAS has a stronger impact on performance than the lower MCAS complexity of our algorithm.

With 98% reads, we observe a similar behavior, with our algorithm outperforming PMwCAS by 2.3× and 1.8× for list sizes 5 and 50 respectively; for list size 500, PMwCAS outperforms our algorithm by 1.29× on average.

5.7.4 B+-tree Benchmark

In this benchmark we operate on a B+-tree which supports search and update (insert and delete) operations. Insertions and deletions use k-CAS, where k may vary, e.g., depending on whether the operation led to nodes being split or merged.

Similar to the previous benchmark, we initialize the B+-tree with a configurable number of

entries; threads then select operations and values at random. We perform the benchmark with three initial tree sizes (16, 512 and 4096) and two operation distributions: 80% reads and 98% reads (as for the previous benchmark, performance graphs for the 50% reads and 100% reads cases are shown in Appendix D.2). As before, the initial size of the tree is half of the key range. Results are shown in Figure 5.3.

We observe a similar behavior to the previous benchmark. With both 80% reads and 98% reads, our algorithm outperforms PMwCAS under high and medium contention (because it performs fewer CASes and triggers less helping) and is slightly outperformed under low contention (where helping no longer plays a major role).

5.8 Related Work

Since the problem of atomic read-modify-write to several locations is central to many concurrent algorithms, significant work on this topic has been done.

Lock- and wait-free implementations of MCAS. Our algorithm shares similarities with previous work [106, 226]: as has become standard practice, it uses operation descriptors and a three-phase design (locking, status-change and unlocking). However, our algorithm introduces key differences with respect to previous work: it defers the unlocking phase and combines it with the reclamation of descriptors, without compromising correctness. This deferment has a triple beneficial effect on complexity: (1) it removes k CASes from the critical path, (2) it allows these CASes to be amortized across several operations, and (3) it removes the onus of ABA-prevention from the locking phase, thus shaving off k further CASes from the latter.

Table 5.1 summarizes the differences between our algorithm and existing non-blocking MCAS implementations. The results in Table 5.1 reflect the number of CASes per MCAS operation required for correctness by each algorithm in the common uncontended case. We note that previous MCAS implementations perform descriptor cleanup immediately after applying MCAS, and it is not clear how to separate cleanup from these algorithms while preserving correctness. If we take the cleanup cost into consideration for our algorithm as well, its theoretical (worst-case) complexity becomes $2k+1$, the same as some of the previous work. As our experiments in Section 5.7 demonstrate, however, the number of CASes in the cleanup phase is negligible in practice. Furthermore, we highlight the fact that unlike most previous work, including the one that employs $2k+1$ CASes, readers in our case do not write into the shared memory in the common case, even when cleanup is considered.

Israeli and Rappoport [124] propose a lock-free and disjoint-access-parallel implementation based on LL/SC and show how LL/SC can be obtained from CAS. Their algorithm requires storing per-thread valid bits at each memory location, thus limiting the number of bits available for data. In the absence of contention, a k-CAS requires $3k+2$ CAS instructions if using

Table 5.1 – Comparison of non-blocking MCAS implementations in terms of the number of CAS instructions required, whether readers perform writes to shared memory or expensive atomic instructions, and the number of persistent fences (all per k-word MCAS, in the uncontended case).

	CASes	Readers write	P. fences
Israeli and Rappoport [124]	$3k+2$	Yes	N/A
Anderson and Moir [11]	$3k+2$	Yes	N/A
Moir [182]	$3k+4$	Yes	N/A
Harris et al. [106]	$3k+1$	No	N/A
Ha and Tsigas [101, 102]	$2k+2$	Yes	N/A
Attiya and Hillel [22]	$6k+2$	N/A	N/A
Sundell [219]	$2k+1$	Yes	N/A
Feldman et al. [88]	$3k-1$	Yes	N/A
Wang et al. [226] (volatile)	$3k+1$	No	N/A
Wang et al. [226] (persistent)	$5k+1$	No	$2k+1$
Our algorithm	$k+1$	No	2

the LL/SC implementation from CAS provided in the paper. In their implementation, uncontended reads (i.e., read operations that do not help concurrent MCAS operations) perform expensive atomic LL instructions, which can be emulated by writes to shared memory, thus limiting performance in common read-heavy workloads.

Anderson and Moir [11] propose a wait-free implementation also based on LL/SC. The strong progress guarantee comes with high space requirements: each memory word needs to be followed contiguously by an auxiliary word containing information needed to help complete an ongoing operation on the memory word.

Moir [182] simplifies [11] considerably by removing the requirement of wait-freedom. Instead, his algorithm is conditionally wait-free: it is lock-free and provides a means to communicate with an external helping mechanism which may cancel MCAS operations that are no longer required to complete.

Harris et al. [106] introduce a lock-free algorithm based on CAS operations. In order to avoid the ABA problem, the algorithm uses a double-compare-single-swap primitive (implemented using two CAS instructions, in the absence of contention) to make each target word point to a global MCAS descriptor while ensuring that the descriptor is still active. In order to distinguish between values and descriptors, the two least-significant bits are reserved in each word. In total, a k-word MCAS uses $3k+1$ CAS instructions in the uncontended case.

Ha and Tsigas [101, 102] provide lock-free algorithms which measure the amount of contention on MCAS target words and react by dynamically choosing the best helping policy.

Attiya and Hillel [22] give a lock-free implementation using CAS and DCAS that requires $6k+2$

CAS instructions for a k-word MCAS in the uncontended case. To avoid the ABA problem, this algorithm stores a tag with each pointer which it atomically increments every time the pointer changes. The algorithm uses a conflict-resolution scheme in which contending operations decide whether to help or reset one another based on how many locations each operation acquired before the conflict was detected (preference is given to operations that own more locations). Their implementation does not provide a separate *read* operation.

Sundell [219] proposes a scheme that uses $2k+1$ CAS instructions for a k-word MCAS (in the absence of contention). An MCAS operation first uses CAS to acquire ownership of each target word, changes the status using a CAS and then uses CAS to write the final values back into the target word. The algorithm is wait-free under the assumption that there is a bound on the number of MCAS operations with equal old and new values.

Feldman et al. [88] propose an algorithm that is both wait-free and ABA-free. In their helping mechanism, a thread actively announces if it is blocked (i.e., if it fails to complete due to concurrent MCAS operations), relying on contending operations to help it to complete.

General techniques. Transactional memory (TM) [114, 214] can be seen as the most general approach to providing atomic access to multiple objects. It allows a block of code to be designated as a transaction and thus executed atomically, with respect to other transactions. Thus, TM is strictly more general than MCAS. This generality comes at a cost: software implementations of transactional memory (STM) have prohibitive performance overheads, whereas hardware support (HTM) is subject to spurious aborts and thus only provides "best-effort" guarantees.

As any concurrent object, MCAS can be implemented using a universal construction [110], but such an implementation is not disjoint-access-parallel and has high overhead.

Restricted and extended multi-word operations. Previous work has explored other operations that atomically read and modify multiple words. These operations are either more general or more restricted than MCAS.

Luchangco, Moir and Shavit [166] present an obstruction-free implementation of a "k-compare-single-swap", which compares on k words but only modifies one word (more restricted than MCAS). Their algorithm is based on LL/SC, for which they give an obstruction-free implementation from CAS.

Brown et al. [44] introduce extensions to LL/SC called LLX/SCX, which are more general than k-compare-single-swap, but more restricted than MCAS. LLX/SCX primitives operate on sets of data records, each comprising several words. SCX allows modifying a single word of a data record, conditional on the fact that no data record in a specified set was modified since LLX was last performed on it. Furthermore, SCX allows finalizing a subset of the data records, preventing them from being modified again. While LLX/SCX and MCAS can be used to solve

similar problems, MCAS is more generic, as it allows modifying k words atomically, whereas LLX/SCX only allow modifying a single word.

Timnat et al. [220] propose an extension of MCAS called MCMS (Multiple Compare Multiple Swap), which also allows addresses to be compared without being swapped (more general than MCAS). They provide implementations of MCMS based on HTM and on the algorithm by Harris et al. [106].

Persistent MCAS. Pavlovic et al. [193] provide an implementation of MCAS for persistent memory which differs from ours in the progress guarantee (theirs is blocking) and hardware assumpions (theirs uses HTM). In their algorithm, a transaction is used to atomically verify expected values and acquire ownership of all target locations. In case of success, the new values are written non-transactionally. Reads that encounter a location owned by an MCAS operation block until the location is no longer owned.

Wang et al. [17, 226] introduce the first lock-free persistent implementation, based on the algorithm of Harris et al. [106]. The main differences with respect to our algorithm are outlined in Table 5.1. This algorithm uses a per-word dirty flag to indicate that the word is not yet guaranteed to be written to persistent memory. Operations encountering a set dirty flag will persist the associated word and then unset the flag. This technique avoids unnecessary persistent flushes, but uses 2 extra CAS instructions per target location in order to manipulate the dirty flag. In total, this algorithm uses $5k+1$ CAS instructions for a k-word MCAS in the uncontended case. Their implementation does not use explicit persistent fences; instead, it relies on the CAS instructions that are already required to unset the dirty flag to also enforce ordering among write backs [123]. Their original algorithm uses $2k+1$ such "CAS-fences", but we believe it can be modified to only require 3 persistent fences.

In our work we use the recent durable linearizability correctness condition [127], which assumes a full-system crash-recovery model, but other models of persistent memory can be explored in this context [6, 30, 67, 100, 196].

5.9 Conclusion

Atomic multi-word primitives significantly simplify concurrent algorithm design, but existing implementations have high overhead. In this chapter, we propose a simple and efficient lock-free algorithm for multi-word compare-and-swap, designed for both volatile and persistent memory. The complementary lower bound shows that the complexity of our algorithm, as measured in the number of CASes in the uncontended case, is nearly optimal.

Memory Reclamation Part III

6 Fast and Robust Memory Reclamation for Concurrent Data Structures

In concurrent systems without automatic garbage collection, it is challenging to determine when it is safe to reclaim memory, especially for lock-free data structures. Existing concurrent memory reclamation schemes are either fast but do not tolerate process delays, robust to delays but with high overhead, or both robust and fast but narrowly applicable. This chapter introduces QSense, a novel concurrent memory reclamation technique. QSense is a hybrid technique with a fast path and a fallback path. In the common case (without process delays), a high-performing memory reclamation scheme is used (fast path). If process delays block memory reclamation through the fast path, a robust fallback path is used to guarantee progress. The fallback path uses hazard pointers, but avoids their notorious need for frequent and expensive memory fences. QSense is widely applicable, as we illustrate through several lock-free data structure algorithms. Our experimental evaluation shows that QSense has an overhead comparable to the fastest memory reclamation techniques, while still tolerating prolonged process delays.

6.1 Introduction

6.1.1 The Problem

Any realistic application requires its data structures to grow and shrink dynamically and hence to reclaim memory that is no longer being used. For the foreseeable future, many high-performing applications, such as operating systems and databases, will be written in languages where programmers manage memory explicitly (such as C or C++). There is thus a clear need for concurrent data structures that scale <u>and</u> efficiently allocate/free memory. Designing such data structures is however challenging, as it is not clear when it is safe to free memory, especially when locks are prohibited (lock-free constructions [90, 115]).

To illustrate the difficulty, consider, as illustrated in Figure 6.1, two processes p_1 and p_2 concurrently accessing a linked list of several nodes. Process p_1 is reading node n_1, while p_2 is concurrently removing node n_1. Assume p_2 unlinks n_1 from the list. Then, p_2 needs to decide

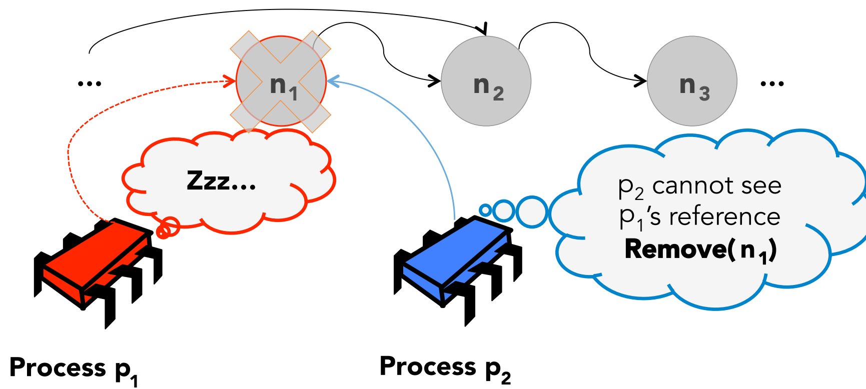


Figure 6.1 – The concurrent memory reclamation problem.

whether n_1's memory can be freed. If p_2 were allowed to block p_1 and inspect p_1's references, then it could easily determine whether freeing n_1 is safe. But in a lock-free context, p_2 has a priori no way of knowing if p_1 is still accessing n_1 or not. So, if p_2 goes ahead and frees node n_1, it triggers an illegal access next time p_1 tries to use n_1.

6.1.2 The Trade-off

Various approaches have been proposed to address the issue of concurrent, programmer-controlled memory reclamation for lock-free data structures. *Hazard pointers* (HP) [179] (which we recall in more details in Section 6.3) is perhaps the most widely-used method. Basically, the programmer publishes the addresses (hazard pointers) of nodes for as long as they cannot be safely reclaimed. A reclaiming process must ensure that a node it is about to free is not marked by any other process. The hazard pointer methodology holds two important advantages: (1) it is wait-free and (2) it is applicable to a wide range of data structures. However, hazard pointers also have a notable drawback: they require a memory fence instruction to be issued for every node traversed in the data structure. This can decrease performance by up to 75% (as we will see in Section 6.7).

Most memory reclamation techniques that seek to overcome the performance penalty of hazard pointers have been analyzed in terms of amortized overhead [4, 40, 43, 107]: the overhead of reclamation operations is spread across several node accesses or across several operations, thus considerably reducing their impact on performance. *Quiescent State Based Reclamation* (QSBR) (which we recall in Section 6.3), is among the most popular schemes applying the amortized overhead principle [43, 107]. QSBR is fast and can be applied to virtually any data structure. However, QSBR is *blocking*: if a process is delayed for a long time (a process failure is a particular type of delay), an unbounded amount of memory might remain unreclaimed. As such, using QSBR with lock-free data structures would negate one of the main advantages of lock-freedom: *robustness* to process delays.

There have indeed been several proposals for achieving both lock-freedom and low amortized overhead [40, 43]. Yet, these are ad-hoc methods that apply only to certain well-chosen data structures. They require significant effort to be adapted to other data structures (as we discuss in Section 6.8).

Overall, the current prevalent solutions for concurrent memory reclamation are either *wait-free* but *with high overhead*, *fast* but *blocking*, or *ad-hoc*, lacking a clear and systematic methodology on how to apply them.

6.1.3 The Contributions

We design, implement and evaluate *QSense*, a novel technique for concurrent memory reclamation that is at the same time easily applicable to a wide range of concurrent data structures (including lock-free ones), robust, and fast.

QSense uses a hybrid approach to provide fast and robust memory reclamation. Figure 6.2 depicts a high-level view of QSense. In the common case (i.e., when processes do not undergo prolonged delays), the fast QSBR scheme is employed. A delay could be caused, for instance, by cache misses, application-related delays, or being descheduled by the operating system. By *prolonged delay*, we refer to a delay of a process p_1 that is long enough such that a large number of nodes (larger than a given configurable threshold) are removed but cannot be safely reclaimed by another concurrent process p_2. If prolonged process delays are detected, QSense automatically switches to a fall-back memory reclamation scheme that is robust. When all processes are active again (no more prolonged delays), the system automatically switches back to the fast path. By *robustness* we mean that any process performing operations on the data structure (called worker process) will finish any action related to memory reclamation within a bounded number of steps, regardless of the progress of other worker processes. To guarantee this progress, we require certain timing assumptions about a set of auxiliary background processes, that do not participate in the actual data structure operations (all assumptions are discussed in Section 6.5).

The fall-back path consists of a subprotocol we call *Cadence*, a novel amortized variant of the widely-used hazard pointer mechanism. Cadence overcomes the necessity for per-node memory barriers during data structure traversal, significantly increasing performance. We achieve this through two new concepts. The first is *rooster processes*: background processes that periodically wake up and generate context switches, which act as memory barriers. The periodic context switches ensure that any hazard pointer becomes visible to other processes within a bounded time T. The second concept is *deferred reclamation*: a process p only reclaims a removed node n after n has been awaiting reclamation for longer than T. This ensures any hazard pointer potentially protecting n must be visible. Therefore, n's memory can be safely freed provided that no hazard pointers are protecting n. Cadence can be used either as part of QSense or as a stand-alone memory reclamation scheme.

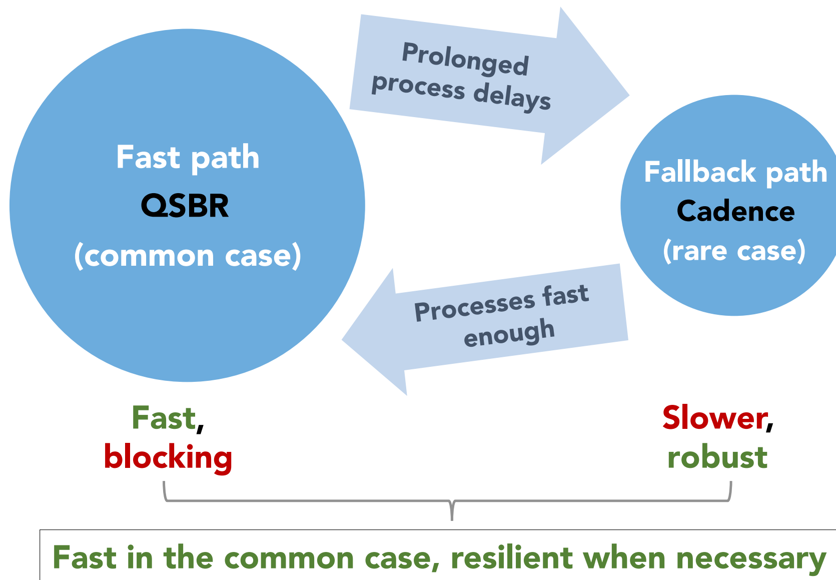


Figure 6.2 – A high-level view of QSense.

QSense requires minimal additions to the data structure code, consisting only of a number of calls to functions provided by the QSense interface. We present a simple set of rules that determine where to place these calls. QSense communicates with data structures through three functions: **`manage_qsense_state`**, **`assign_HP`** and **`free_node_later`**. The rules concerning where to call these three functions are:

1. call **`manage_qsense_state`** in states where no references to shared objects are held by the processes (usually in between data structure operations).
2. call **`assign_HP`** before using a reference to a node so as to inform other processes that the node's memory should not be reclaimed yet.
3. call **`free_node_later`** whenever a node is removed from the data structure (where **`free`** would be called in a sequential setting).

We show empirically that QSense achieves good performance in a wide range of scenarios, while providing the same progress guarantees as a hazard pointer based scheme. Our experiments in Section 6.7 show that QSense achieves at most 29% overhead on average over leaky implementations (i.e., in which memory in never reclaimed) of a lock-free concurrent linked list, a skip list and a binary search tree. Moreover, QSense outperforms the popular hazard pointers technique by two to three times. To illustrate this point, Figure 6.3 shows sample results from our experiments, comparing QSense to hazard pointers and no memory reclamation (leaky implementation) on a concurrent linked list.

Roadmap. The rest of this chapter is organized as follows. In Section 6.2, we pose the problem.

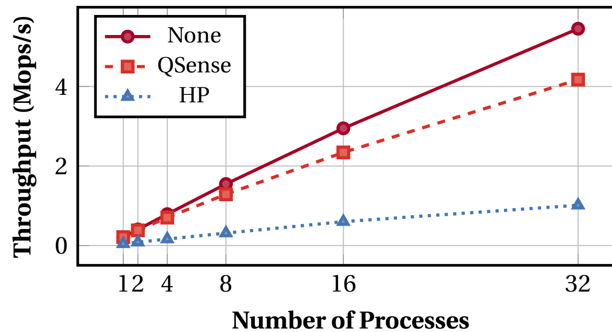


Figure 6.3 – QSense, HP and no reclamation on a linked list of 2000 elements, with a 10% updates workload.

In Section 6.3, we recall prior work that inspired QSense. In Section 6.4, we give an overview of QSense. Then, in Section 6.5, we dive into the details of Cadence and detail the assumptions needed for its correctness. In Section 6.6, we prove safety and liveness properties of QSense. In Section 6.7, we compare QSense's performance against that of popular memory reclamation schemes. Finally, in Section 6.8, we describe related work and we conclude in Section 6.9.

6.2 Model and Problem Definition

We consider a set of n processes that communicate through a set of shared memory locations using primitive memory access operations. A *node* is a set of memory locations that can be viewed as a logical entity. A *data structure* consists of one or more fixed nodes that can always be accessed directly by the processes, called *roots*, and the set of nodes that can be reached by following pointers from the roots.

6.2.1 Node States

At any given time, a node can be in one of five states [179]: (1) *Allocated* — the node has been allocated by a process, but not yet inserted into the data structure. (2) *Reachable* — the node is reachable by following valid pointers from the roots of the data structure. (3) *Removed* — the node is no longer *reachable*, but may still be in use by some processes. (4) *Retired* — the node is removed and cannot be used by any process, but is not yet *free*. (5) *Free* — the node's memory is available for allocation.

6.2.2 The Memory Reclamation Problem

We can now state the *memory reclamation problem* as follows: given some *removed* nodes, make them available for re-allocation (i.e., change their state to *free*) after it is no longer possible for any process (except the reclaiming process) to access them (i.e., after they have become *retired*).

The problem is distinct from garbage collection. Here, we are only concerned with the reclamation of memory used by data structure nodes, and not the reclamation of arbitrary memory regions. Moreover, the nodes whose memory needs to be reclaimed are expressly marked by the programmer after they have been explicitly unlinked from the data structure and are no longer reachable.

6.2.3 Terminology

Safe: A node n is *safe* [179] for a process p if: (1) n is *allocated* and p is the process that allocated it, or (2) n is *reachable*, or (3) n is *removed* or *retired* and p is the process that removed n from the data structure.

Possibly unsafe: A node n is *possibly unsafe* [179] for a processes p if it is impossible, using only knowledge of p's private variables and the semantics of the algorithm, to positively establish that n is *safe* for p. Informally, a node n is *possibly unsafe* if it is impossible to determine using only local process data and knowledge of the algorithm whether accessing n will not trigger an access violation (e.g., if n has been reclaimed in the meanwhile).

Access hazard: An *access hazard* [179] is a step in the algorithm that might result in accessing a *possibly unsafe* node for the process that is executing the algorithm.

Hazardous reference: A process p holds a *hazardous reference* [179] to a node n if one of p's private variables holds n's address and p will reach an *access hazard* that uses n's address. Informally, a *hazardous reference* is an address that will be used later in a hazardous manner (i.e., to access *possibly unsafe* memory) without further verification of safety.

6.3 Background

In this section we recall quiescent state based reclamation and hazard pointers, the techniques QSense builds upon.

6.3.1 Quiescent State Based Reclamation

Quiescent State Based Reclamation (QSBR) is a technique which emerged in the context of memory management in operating system kernels [16]. Quiescence-based schemes are fast, outrunning popular pointer-based schemes under a variety of workloads [107]. Therefore, in QSense, we chose a quiescence technique for the fast path. Though not a contribution of this chapter, the QSBR technique is detailed below, for completeness.

QSBR makes use of *quiescent states* (at process level) and *grace periods* (at system level).

A process is in a *quiescent state* if it does not hold references to any shared objects in the data

structure. Quiescent states need to be specified at the application level. Typically, a process is in a quiescent state whenever it is in between operations (read/insert/delete). In practice, quiescent states are declared after processes have finished a larger number of operations — called the *quiescence threshold* — as batching operations in this way boosts performance.

A *grace period* is a time interval in the execution during which each worker process in the system goes through at least one quiescent state. If the time interval $[a, b]$ is a grace period, after time b no process holds hazardous references to nodes that were removed before time a. The occurrence of grace periods is managed through an epoch-based technique [43, 107]. At every step of the execution, every process is in one of three logical epochs. Each process has three lists in which removed nodes are stored, called *limbo lists* (one per epoch). If a node n has been removed when a process p was in epoch i, n will be added to p's i^{th} limbo list. Each process keeps track of its local epoch and all processes have access to a shared global epoch. When a process p declares a quiescent state, it does the following. If p's local epoch e_p is different than the global epoch e_G, then p updates e_p to e_G. Else, if all processes have their local epoch equal to e_G (including p), p increments e_G by 1.

The Problem of Robustness in QSBR. The main advantage of QSBR is the low overhead. Nevertheless, its lack of robustness to significant process delays makes its out of the box use unsuitable for some practical applications. As we show in Section 6.7, if achieving a grace period is no longer possible or takes a significant amount of time, the system might run out of memory and eventually block. In Section 6.5, we show how we address the problem of resilience to prolonged process delays in QSense.

6.3.2 Hazard Pointers (HP)

The main idea behind the hazard pointers scheme [179] is to associate with each process a number of single-writer multi-reader pointers. These pointers — called *hazard pointers* — are used by processes to indicate which nodes they might be about to access without further validation. The nodes marked by hazard pointers are unsafe to reclaim.

The hazard pointer scheme mainly consists of *node marking* and *node reclamation*. *Node marking* is the assignment of hazard pointers to nodes. This ensures that the reclaiming process can discern which of the nodes that were removed from the data structure are *retired* (and thus safe to reclaim). *Node reclamation* is a procedure that makes nodes available for re-allocation. Every time a process removes a node from a data structure, the process adds a reference to the node in a local list of removed nodes. After a given number of node removals, each process will go through its list of removed nodes and will free those nodes that are not protected by any hazard pointers. The programmer needs to ensure the following condition:

Condition 6.3.1. *At the time of any hazardous access by a process p to the memory location of a node n (access hazard), n has been continuously protected by one of p's hazard pointers since a time when n was definitely safe for p.*

Michael [179] provides a methodology for programmers to enforce this condition: (1) Identify all hazardous references and access hazards in the data structure code. (2) For each hazardous reference, determine the step when the reference is created and the last access hazard where the reference is used. This is the period when the reference needs to be protected by a hazard pointer. (3) Examine the overlap of the periods from step 2. The maximum number of hazard pointers is the maximum number of distinct hazardous references that exist simultaneously for the same process. (4) For each hazardous reference, assign a hazard pointer to it and immediately afterwards, verify if the reference (the node) is still safe. If the verification fails, follow the path of the original algorithm that corresponds to failure due to contention (e.g., try again, backoff etc.).

The Problem of Instruction Reordering in HP. An important practical consideration when applying the above methodology is instruction reordering [115, 170]. In most modern processors, instructions may be executed out of order for performance considerations. In particular, in the widespread x86/AMD 64/SPARC TSO memory models, stores may be executed after loads, even if the stores occur before loads in the program code. Instruction reordering is relevant in the case of hazard pointers due to step 4 in the methodology above. We assign a hazard pointer and then verify that the node is still safe, thus ensuring that the hazard pointer starts protecting that node from a time when the node is definitely safe. However, if assigning the hazard pointer (a store) is reordered after the validation (a load), then we can no longer be certain that the node is still safe when the hazard pointer becomes visible to other processes. Therefore, it is necessary for the programmer to insert a *memory barrier* between the hazard pointer assignment and the validation. Listing 6.1 shows the high-level instructions that need to be added to the data structure code when accessing a node, if hazard pointers are used (lines 2–4).

A *memory barrier* [115] (or fence) is an instruction that ensures no memory operation is reordered around it — in particular, all stores that occur before a memory barrier in program order will be visible to other processes before any loads appearing after the barrier is executed. Therefore, when validating that a node is still safe (as per step 4 in the methodology), we can be certain that the hazard pointer is already visible.

Listing 6.1 – High-level steps taken when accessing a node in a data structure using *hazard pointers.*

```
1 Read reference to node n
2 Assign a hazard pointer to n
3 Perform a memory barrier
4 Check if node n is still valid
5 Access n's memory
6 Release reference to n (e.g., move to successor node)
```

Memory barriers are expensive instructions. They can take hundreds of processor cycles. This cost results in a significant performance overhead for hazard pointer implementations, especially in read-only data structure operations (update operations typically use other expensive synchronization primitives such as compare-and-swap, so the marginal cost of memory barriers due to hazard pointers is much lower than for read-only operations). Moreover, these memory barriers must be performed on a per-element basis, which causes the performance of hazard pointers to scale poorly with data structure size.

6.4 An Overview of QSense

Combining QSBR's high-performance with hazard pointers' robustness in a hybrid memory reclamation scheme is appealing. In this section, we first argue why merging QSBR with the original hazard pointers technique is also however a challenge. Then, we give a high-level view of QSense.

6.4.1 Rationale

One could imagine a hybrid scheme where QSBR and hazard pointers are two separate entities, with the switch between the two schemes triggered by a *signal* or *flag*. QSBR would run in the common case (when no process delays are observed) and hazard pointers would be employed when a long process delay makes quiescence impossible. However, after a switch to hazard pointers based reclamation, hazardous references from when the system was running in QSBR mode would need to be protected as well. So, hazard pointers should be protecting nodes during the entire execution of the system, regardless of the mode of operation the system is currently in. As discussed above, the original hazard pointers algorithm requires a memory barrier call after every hazard pointer update, to ensure correctness. However, ideally, when the system operates in QSBR mode, the per-node memory barriers required by hazard pointers should be eliminated. Per-node memory barriers should be placed as specified in the HP algorithm only when the system goes into fallback mode. Nonetheless, such an approach is not correct. The scenario in Listing 6.2 illustrates why.

Consider two processes, P_R and P_D, at a time t when the system makes the switch from the fast path to the fallback path. P_R is a reader process performing steps R_1 to R_5 and P_D is a deleting process performing steps D_1 to D_4, during which it detects that it must switch to the fallback scheme.

Assume that P_D's steps are not reordered (we can use memory barriers to ensure this, since we are mainly concerned with removing memory barriers from read-only operations, but not necessarily from deletion operations). However, P_R's steps can be reordered; more precisely, in the TSO model, the R_2 store can be delayed past all subsequent reads [170] if P_R does not detect that the fallback-flag is turned on and does not perform a memory barrier.

Listing 6.2 – Example of illegal operation interleaving.

```
(R1). Read a pointer to a node n (Load)
(R2). Assign a hazard pointer to n (Store)
(R3). If fallback mode is active (Load) execute a memory barrier (here, suppose
    ↪ fallback mode is inactive and the memory barrier is not executed)
(R4). Recheck n (Load)
(R5). Use n (Loads and Stores)

(D1). Remove n
(D2). Check fallback−flag (here, suppose fallback mode was activated)
(D3). Scan hazard pointers
(D4). Free n (assuming no hazard pointer points to it)
```

Next, consider the following interleaving of P_R's and P_D's steps. Initially the fallback-flag is off (the system is running in the fast path). The reader will read a reference to n (R_1) and assign a hazard pointer to n (R_2). At R_3 the fallback-flag is off, so the memory barrier is not executed. Thus, the store of the hazard pointer (R_2) can be delayed past all subsequent reads. Then, P_R rechecks n (R_4) and finds that it is still a valid node. Now, suppose that another process P activates the fallback path. This is where P_D steps in and executes D_1 through D_4. Since P_R's store of the hazard pointer was delayed, P_D is free to reclaim the node in question. Finally, in step R_5, P_R will try to use the reference to n that it had acquired without publishing the hazard pointer via a memory barrier and thus attempt to access a reclaimed node, which is incorrect.

If a memory barrier was called after the update of each hazard pointer in both the fast and fallback paths, the QSBR/HP hybrid would function correctly. However, adding per-node memory barriers when running the fast path means re-introducing the main performance bottleneck of the fallback scheme into the fast path. Consequently, the performance of the hybrid in QSBR mode would be similar to its performance in HP mode, what we initially set out to avoid.

The challenge is to eliminate the traversal memory barriers when the system operates in the fast path (QSBR), while optimistically updating the hazard pointer values. We address this challenge by designing Cadence, a hazard pointer inspired memory reclamation scheme which does not require per-node memory barriers upon traversal. Cadence is presented in Section 6.5. Then, in Section 6.6 we show that Cadence is a good candidate for the fallback scheme in QSense, preserving the safety properties of the algorithm, while not hindering the performance of QSBR in the fast path.

6.4.2 QSense in a Nutshell

QSense is a hybrid scheme, unifying the high-performing approach provided by QSBR and the robustness provided by hazard pointers. QSense is an adaptive scheme, using two paths (a

fast path and a fallback path) and automatically deciding when it should switch paths. QSBR constitutes the fast path. QSBR is used in the common case, when all processes are active in the system. As QSBR has been presented in prior work [43, 107], we omit the implementation details.

When one or more processes experience prolonged delays (e.g., blocked in I/O), there is a risk of exhausting the available memory, because quiescence is not possible. In this case, Cadence serves as a safety net. Cadence guarantees that QSense continues to function within a bounded amount of memory. Cadence eliminates the expensive per-node memory fences needed for data structure traversal, the main drawback of the original hazard pointer scheme. Instead of using memory barriers, Cadence forces periodic context switches to make outstanding hazard pointer writes visible. Since the cost of expensive operations (in our case context switches) is spread across a large number of operations, Cadence achieves scalability that is up to three times as good as the original hazard pointer scheme, while maintaining the same safety guarantees. Moreover, using Cadence as the fallback path allows the elimination of memory barriers when running in the fast path. QSense automatically detects the need to switch to the fallback scheme and triggers the switch through a shared flag. Similarly, when all the processes become active again in the system (e.g., return from a routine incurring a longer delay), QSense re-establishes the quiescence-based reclamation mechanism automatically.

Applicability. QSense can be used with any data structure for which both the fast path and the slow path are applicable. Since Cadence does not introduce any additional usability constraints compared to hazard pointers, and QSBR can be applied to virtually any data structure, this means that QSense can be used with any data structure for which hazard pointers are applicable. Applying QSense to a data structure is done in three steps: (1) Call `manage_qsense_state` to declare a quiescent state (e.g., between every two data structure operations). The function automatically handles amortizing overhead by executing the memory reclamation code only once every Q calls to `manage_qsense_state` (where Q is the quiescence threshold introduced in Section 6.3.1). (2) Following the methodology from Section 6.3.2, protect hazardous references by calling `assign_HP`. (3) To reclaim memory, call `free_node_later` when free would be called in a sequential setting. An example of how to apply QSense to a concurrent linked list can be found in Appendix E.2.

6.5 Cadence

In this section, we present Cadence, our fallback path for QSense. We first describe Cadence as a stand-alone memory reclamation technique and then show how we integrate Cadence with QSBR in our QSense scheme.

6.5.1 The Fallback Path

Cadence builds upon hazard pointers, eliminating the need to use per-node memory barriers when traversing the data structure. Note that while Cadence is the fallback path in QSense, it could also be used as a stand alone memory reclamation scheme. Algorithm 6.3 presents the pseudocode implementation of the main functions of Cadence. This scheme is based on two new concepts: *rooster processes* and *deferred reclamation.*

1. **Rooster processes** are a mechanism used to ensure that all new hazard pointer writes become globally visible after at most a period of time T (the sleep interval, a configurable parameter). For every core or hardware context on which a worker process is running, we create a rooster process and pin it to run on that core. Every rooster process has the task of sleeping for a predetermined amount of time T , waking up, and going back to sleep again, in an infinite loop. In this way, every worker process is guaranteed to be switched out periodically, and thus the worker processes' outstanding writes, *including hazard pointers,* will become visible to the other processes (as detailed in *Note on assumptions* below). Therefore, for every time t, all hazard pointers that were published before time $t - T$ are visible to all processes.

2. **Deferred reclamation.** Figure 6.4 illustrates the main idea of how rooster processes and deferred reclamation work together. By verifying that a node n is *old enough* (pseudocode shown in Algorithm 6.3, lines 35–39), the reclaiming process makes sure that at least one rooster process wake-up has occurred since n was removed from the data structure and thus that any hazard pointers protecting n have become visible. If n is removed by a process at time t_0, then at time $t > t_0 + T$, any potential hazard pointers protecting n are visible. This is because these hazard pointers were written before t_0 (by the methodology in Section 6.3.2) and at least one rooster process wake-up has occurred between t_0 and t. Therefore, at t, the reclaiming process can safely free the node's memory provided that no hazard pointers are protecting it.

When a node n is removed from the data structure (when the `free` function would be called in a sequential setting), n is *timestamped* and placed inside the removing process's local list of nodes awaiting reclamation. Once every R such node removals, a `scan` of the list is performed (see Algorithm 6.3, lines 14–33). A `scan` inspects the process's removed nodes list and frees those nodes that are safe to reclaim (retired). Nodes that are **old enough and are not protected by any hazard pointers** are freed; hazard pointers of all the worker processes are checked, not only the reclaiming process's. The rest of the nodes — which are either not old enough, or are protected — are left in the removed nodes list, to be reclaimed at a later time (Algorithm 6.3, lines 26–29).

From the programmer's perspective, using Cadence is essentially identical to using hazard pointers. The difference is that no memory barriers are needed after publishing a new hazard pointer, thus making Cadence up to three times as fast than the original hazard pointers (as

Algorithm 6.3 – Main functions of Cadence.

```
// *** Cadence ***
timestamped_node {
    node* actual_node;
    timestamp time_created;
    timestamped_node* next;};
HP_array HP; // Shared HP array

void assign_HP(node* target, int HP_index) {
    // Assign hazard pointer to target node.
    HP[HP_index] = target;
    // No need for a memory barrier here.
}

void scan(timestamped_node* removed_nodes_list) {
    // Insert non-null values in HP_in_use.
    HP_array HP_in_use = get_protected_nodes(HP);
    // Free non-hazardous nodes
    timestamped_node* tmplist;
    tmplist = removed_nodes_list;
    removed_nodes_list = NULL;
    timestamped_node* cur_node;
    while (tmplist != NULL) {
        cur_node = tmplist;
        tmplist = tmplist->next;
        // Deferred reclamation
        if (!is_old_enough(cur_node) ||
        HP_in_use.find(cur_node->actual_node)) {
            cur_node->next = removed_nodes_list;
            removed_nodes_list = cur_node;
        } else {
            free(cur_node->actual_node);
            free(cur_node);
} } }

boolean is_old_enough(timestamped_node* wrapper_node) {
    current_time = get_current_time();
    age = current_time - wrapper_node->time_created;
    return (age ≥ (ROOSTER_SLEEP_INTERVAL + EPSILON));
}
```

we will see in Section 6.7). Listing 6.4 shows the steps taken when accessing a node, when Cadence is used.

Note on assumptions. For correctness, Cadence relies on the following assumptions. First, we assume that the only instruction reorderings possible are the ones between loads and subsequent reads, as in the TSO model. We assume this in Section 6.3.2 to determine the memory barrier placement and in Section 6.5.1 to justify that the memory barrier is no longer needed.

Second, we require that a context switch implies a memory barrier for the process being

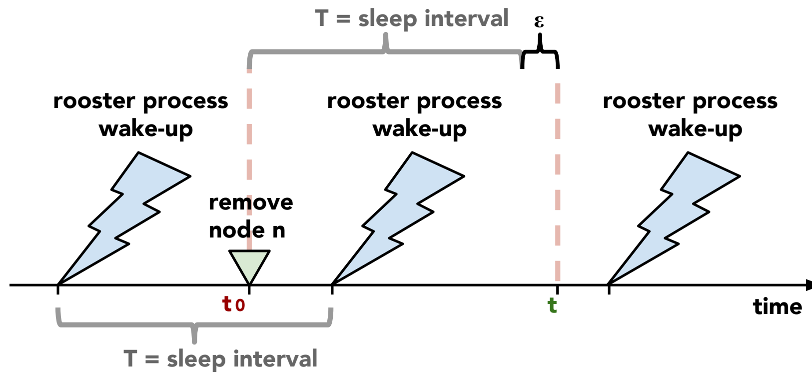


Figure 6.4 – Rooster processes and deferred reclamation.

Listing 6.4 – High-level steps taken when accessing a node in a data structure using *Cadence*.

1. Read reference to node n
2. **Assign a hazard pointer to n**
3. ~~**Perform a memory barrier**~~
4. **Check if n is still valid**
5. Access n's memory
6. Release reference to n (e.g., move to successor node)

switched out. This is necessary to guarantee that hazard pointers published by a process p become visible at the latest right after p is switched out by a rooster process. The low-level locking required to perform a context switch automatically provides a memory barrier for the process being switched out. Although this property is architecture dependent and might not be generally true if architectures change in the future, it does hold for most modern architectures [144, 170].

Third, we assume that rooster processes never fail. This is a reasonable assumption, considering that rooster processes do not take any steps that could produce exceptions: their only actions are going to sleep and waking up. However, small timing inconsistencies might appear, in the form of (1) rooster processes possibly taking slightly longer than T between wake-ups, i.e., “oversleeping" (it is reasonable to assume that this difference is small, since modern operating systems use fair schedulers [165]) and (2) different cores seeing slightly different times when creating and comparing timestamps. We make the assumption that these inconsistencies are bounded and introduce a tolerance ϵ to account for them. We use ϵ explicitly, as shown in Figure 6.4: to verify if a removed node n is old enough, we compute the difference between the current value of the system clock and n’s timestamp. If the time difference is larger than $T+\epsilon$, then n is old enough.

Note that we are making no assumptions about the *worker processes* (i.e., the processes performing read or write operations on the data structure). In particular, they may be delayed

for an arbitrary amount of time. Therefore, the model under which our construction is correct and wait-free is partly asynchronous (the worker processes) and partly synchronous (the rooster processes).

6.5.2 Merging the Fast and Fallback Paths

In QSense, from a high-level design point of view, the fast path (QSBR) and the fallback path (Cadence) can be viewed as two separate entities, functioning independently. The switch between the two modes is triggered via a shared flag, called the *fallback-flag*. However, even if the two modes of operation are logically distinct, there are elements of the two schemes that have been merged or that are continuously active. Even if QSense is running in the fast path, hazard pointers still have to be set. As explained in Section 6.4.1, this is necessary because in the case of a switch to the fallback path, hazardous references need to be protected. Furthermore, for similar reasons, timestamps need to be recorded when a node is removed, regardless of the mode of operation of QSense. Moreover, when running in fallback mode and performing a **`scan`**, QSBR's **`limbo_list`** (with all three epochs) becomes the **`removed_nodes_list`** scanned by Cadence. Pseudocode for the main functions used by QSense is shown in Algorithms 6.5 and 6.6 (unless stated otherwise, all pseudocode references in the rest of this section refer to Algorithms 6.5 and 6.6). An example of how to apply QSense to a concurrent linked list can be found in Appendix E.2.

Any of the worker processes can trigger the switch between the fast and fallback paths. The switch can be split into the following sub-problems:

1. **Detecting the need to switch to the fallback path.** The switch from QSBR to Cadence is triggered when a process detects that its removed (but not freed) nodes list reached a size C, where C is a parameter of the system (lines 53–59). Reaching a large removed nodes list size for one process indicates that quiescence was not possible for an extended period.
2. **Switching from the fast path to the fallback path.** QSense signals the switch from QSBR to Cadence through the *fallback-flag*. The process that has detected the need for the switch sets the shared *fallback-flag*. The *fallback-flag* is checked by all processes when performing node reclamation (i.e., calling the **`free_node_later`** function, shown in Algortihm 6.6), so the path switch will eventually be detected by all active processes (line 41). If the flag is set to *fallback mode*, a hazard pointer style **`scan`** as described in Section 6.5.1 is immediately performed to reclaim nodes (lines 42–47; **`scan`** shown in Algorithm 6.3).
3. **Detecting when it is safe to switch back to the fast path.** To determine when to switch from the fallback path to the fast path we need to verify if all processes have once again become active in the system. While the system operates in fallback mode, there is at least one process which cannot participate, because using the fallback path implies that

Algorithm 6.5 – Main QSense functions (I).

```
1  //*** QSENSE interface ***
2  void manage_qsense_state();
3  void assign_HP(node* target, int HP_index);
4  void free_node_later(node* n);

6  // One limbo list per epoch
7  timestamped_node* limbo_list[3];
8  int call_count = 0;
9  int free_node_later_call_count = 0;

11 // *** QSENSE main functions***
12 void manage_qsense_state(){
13     // Batch operations
14     call_count += 1;
15     if (call_count % QUIESCENCE_THRESHOLD != 0) {
16         return;
17     }
18     // Signal that the process is active
19     is_active(process_id);
20     seen_fallback_flag = fallback_flag;
21     if (seen_fallback_flag == FAST_PATH) {
22         // Common case: run the fast path
23         quiescent_state();
24         prev_seen_fallback_flag = FAST_PATH;
25     } else if (seen_fallback_flag == FALLBACK_PATH) {
26         // Try to switch to fast path
27         if ( all_processes_active() ) {
28             // Trigger switch to the fast path
29             fallback_flag = FAST_PATH;
30             prev_seen_fallback_flag = FAST_PATH;
31             quiescent_state();
32         }
33         prev_seen_fallback_flag = FALLBACK_PATH;
34 }   }
```

one of the processes was delayed and quiescence was not possible. To assess whether all the processes have become active in the meantime, we keep an array of *presence-flags* (one flag per worker process), which is reset periodically. After each operation (or batch of operations) on the data structure, processes set their corresponding *presence-flags* to true, to signal that they are active (line 19). Then, processes scan the *presence-flag* array (line 27). If one of the processes sees all of the presence flags set to true, it infers that all processes might be active again in the system and a switch from fallback path to fast path is attempted.

4. **Switching from the fallback path to the fast path.** If QSense runs in fallback mode, but all processes have become present in the meantime, the possibility to switch from Cadence back to QSBR is detected, as described above. Similarly to switching from the fast path to the fallback path, the switch in the opposite direction is signaled through setting the value of the shared *fallback-flag* and immediately declaring a quiescent state

Algorithm 6.6 – Main QSense functions (II).

```
35 void free_node_later (node* n) {
36     // Create timestamped wrapper node
37     timestamped_node* wrapper_node = alloc(size(timestamped_node));
38     wrapper_node->actual_node = n;
39     wrapper_node->time_created = get_current_time();
40     limbo_list[my_current_epoch].add(wrapper_node);
41     seen_fallback_flag = fallback_flag;
42     if (seen_fallback_flag == FALLBACK_PATH &&
43     ++ free_node_later_call_count % R == 0)) {
44         // Running in fallback mode. All three epochs in limbo list are
             ↪ scanned.
45         scan(limbo_list[0]); scan(limbo_list[1]);
46         scan(limbo_list[2]);
47         prev_seen_fallback_flag = FALLBACK_PATH;
48     } else if ( prev_seen_fallback_flag == FALLBACK_PATH &&
49         seen_fallback_flag == FAST_PATH) {
50         // QSBR mode switch triggered by another process.
51         quiescent_state();
52         prev_seen_fallback_flag = FAST_PATH;
53     } else if (size(limbo_list) ≥ C &&
54     prev_seen_fallback_flag == FAST_PATH) {
55         // Trigger switch to fallback mode:
56         fallback_flag = FALLBACK_PATH;
57         prev_seen_fallback_flag = FALLBACK_PATH;
58         scan(limbo_list[0]); scan(limbo_list[1]);
59         scan(limbo_list[2]);
60 }   }
```

(lines 28–31). If the switch to the fast path was already triggered by another process, the new value of the *fallback-flag* will be seen upon retiring a node (in the **free_node_later** function, lines 48–52).

The current version of QSense does not support dynamic membership: processes cannot join or leave the system as an algorithm is running. Also, if a process crashes and never recovers, QSense will switch to fallback mode and stay there forever. Both of these issues can be addressed by adding mechanisms for processes to announce entering or leaving the system and for evicting participating processes that have not quiesced in a long time. We leave these extensions for future work.

6.6 Correctness & Complexity

In this section we argue for the safety and liveness of Cadence and QSense. For completeness, safety and liveness proofs for QSBR can be found in Appendix E.1.

6.6.1 Cadence

Property 6.6.1 (Safety)**.** *If at time t, a node n is identified in the* **scan** *function as eligible to be reused by process p, then no process $q \neq p$ holds a hazardous reference to n at time t.*

Proof. Assume by contradiction that (1) n is identified as eligible for reuse by p at time t and (2) there exists another process q that holds a hazardous reference to n at t. Then by (1) and the **scan** algorithm, at time t, p inspects n's timestamp and finds that n is old enough, meaning that n has been removed from the data structure at a time $t' \leq t - T$ (where T is the rooster process sleep interval, including the tolerance ϵ). Therefore, by Condition 6.3.1 in Section 6.3.2, q has had a hazard pointer hp dedicated to n since a time $t'' \leq t'$. Since $t - t'' \geq T$, the write by q to hp at t'' is visible to p at time t. Therefore, by the **scan** algorithm, p will not identify n as eligible for reuse (since it is protected by a hazard pointer). □

Lemma 6.6.2. *For any process p, at the end of a call to* **scan** *by p, p can have at most $NK + T$ retired nodes in its removed nodes list, where N is the number of processes, K is the number of hazard pointers per process and T is the rooster process sleep interval .*

Proof. At the time of the **scan**, there can be at most NK nodes protected by hazard pointers (since NK is the total number of hazard pointers), and there can be at most T nodes that are not yet old enough (for clarity, we assume that p can remove at most one node per time unit). □

Property 6.6.3 (Liveness)**.** *At any time, there are at most $N(K + T + R)$ retired nodes in the system, where R is the number of nodes a process can remove before it invokes scan.*

Proof. Fix a process p and examine how many nodes can be removed by p but not yet reclaimed. Using Lemma 1, it follows that between two **scan** calls, the number of nodes in p's removed nodes list can grow up to $NK + T + R$, before the next **scan** call is triggered, lowering the size of the removed nodes list to $NK + T$ again. So the maximum size of a process' removed nodes list is $NK + T + R$, where the NK term comes from the nodes that are protected by hazard pointers. When considering the entire system, we can have at most NK HP-protected nodes, and at most $N(T + R)$ non-HP-protected nodes, so the maximum number of retired nodes is $N(K + T + R)$. □

6.6.2 QSense

Property 6.6.4 (Safety)**.** *If a node n is identified at time t by process p as eligible for reuse, then no process $q \neq p$ holds a hazardous reference to n.*

Proof. Since all processes keep track of both hazard pointers (exactly as in Cadence) and of epochs (exactly as in QSBR), regardless of whether the system is in the fast path or in the

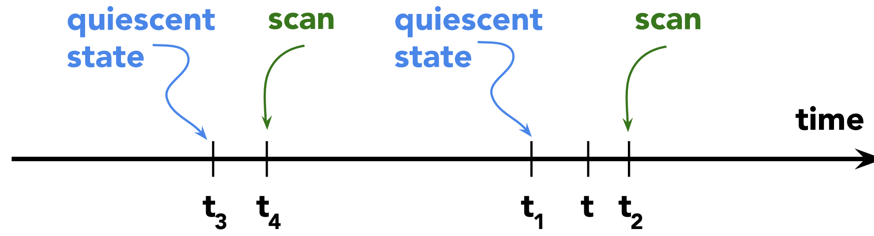


Figure 6.5 – Time diagram for the proof of QSense liveness.

fallback path, the safety guarantees of both methods are maintained. □

For the next property, we define a *legal* value of C, the threshold introduced in step 1 of Section 6.5.2. C refers to the size of the removed nodes list and is used for determining when to switch to the fallback path. We say that C is legal if $C > \max(mQ, NK+T, (K+T+R)/2)$, where Q is the quiescence threshold introduced in Section 6.3, m is the maximum number of nodes that can be removed by a single operation, and N, K and T are as in Section 6.6.1. Picking a legal value for C is always possible since C is a configurable parameter.

Property 6.6.5 (Liveness). *If C has a legal value then, at any time, there can be no more than $2NC$ retired nodes in the system.*

Proof. Assume by contradiction that there is a time t when there are $U > 2NC$ retired nodes in the system. Then there exists a process p such that at time t, p has more than $2C$ retired nodes.

Let $t_1 < t$ be the time of the last quiescent state called by p before t (as illustrated in Figure 6.5). Since t_1, p has completed at most Q operations (otherwise p would have gone through another quiescent state before t) and has therefore removed at most mQ nodes. Therefore, at t_1, p has at least $2C - mQ > C$ (using the fact that $C > mQ$) retired nodes and therefore triggers a switch to the fallback path. This means that p will call a hazard pointers **scan** before starting any other operation, by construction of the QSense algorithm (step 2 in Section 6.5.2). Let t_2 be the time of this first **scan** after t_1. After this scan is complete, p can have at most $NK + T$ retired nodes, by Lemma 1. If $t > t_2$, then at t, p can have at most $NK + T + mQ < 2C$ retired nodes ($NK + T$ at most at the end of the **scan** plus mQ because p has completed at most Q operations since t_2), a contradiction. Therefore it must be the case that $t_1 < t < t_2$. Since the **scan** at t_2 is called immediately after the quiescent state at t_1, without other operations being started by p (by step 2 in Section 6.5.2), the number of retired nodes does not increase between t_1 and t. So at t_1, p has more than $2C$ retired nodes. We now show this to be impossible.

Let t_3 be the time of the last quiescent state called by p before t_1. Since p performed Q operations between t_3 and t_1, it follows that at t_3, p had at least $2C - mQ > C$ retired nodes, thus triggering a switch to the fallback state and a **scan** at time t_4, $t_3 < t_4 < t_1$. After this **scan**, p had at most $NK + T$ retired nodes and therefore, at t_1, p had at most $NK + T + mQ < 2C$

retired nodes, a contradiction because we had shown that p had more than $2C$ retired nodes at t_1. This completes the proof. □

6.7 Experimental Evaluation

We first describe the experimental setup of our evaluation. We proceed with presenting the methodology of our experiments and we finally discuss our evaluation results.

6.7.1 Experimental Setting

We apply QSense to a lock-free linked list [177], a lock-free skip list [90] and a binary search tree [184]. The base implementations of these data structures are taken from ASCYLIB [74]. The code to reproduce our experiments is available at https://github.com/zablotchi/qsense. We compare the performance of QSense to that of QSBR, HP and a leaky implementation (i.e., in which memory is never reclaimed). Our evaluation was performed on a 48 core AMD Opteron with four 12-core 2.1 GHz Processors and 128 GB of RAM, running Ubuntu 14.04. Our code was compiled with GCC 4.8.4 and the `-O3` optimization level.

Each experiment consists of a number of processes concurrently performing data structure operations (searching, inserting or deleting keys) for a predefined amount of time. Each operation is chosen at random, according to a given probability distribution, with a randomly chosen key. An initialization is performed before each experiment, where one process fills the data structure up to half the key-range.

6.7.2 Methodology

The purpose of our evaluation is two-fold. First, we aim to determine whether QSense performs similarly to QSBR in the base case. It is important to highlight the base case behavior of QSense (i.e., when no processes undergo prolonged delays), since this is the expected execution path in most scenarios. To this end, we run a set of tests emphasizing the scalability with respect to the number of cores. For this first category of tests, the system throughput is recorded as a function *of the number of cores* (each process is pinned to a different core). The number of cores is varied from 1 to 32, the operation distribution is fixed at 50% reads and 50% updates (i.e., 25% inserts, 25% deletes) and the key range sizes are fixed at 2000 for the linked list, 20000 for the skip list and 2000000 for the binary search tree.

The second goal of the evaluation is to examine the behavior of QSense in case of prolonged process delays. To this end, we run a set of tests that include periodic process disruptions and measure the system throughput as a function of *time*. We observe the switch from QSBR to Cadence when the system senses that one of the processes was delayed and the switch back to QSBR when all the processes became active in the system again. We seek to trigger a switch between the paths (QSBR to Cadence or oppositely) every 10 seconds. To induce the

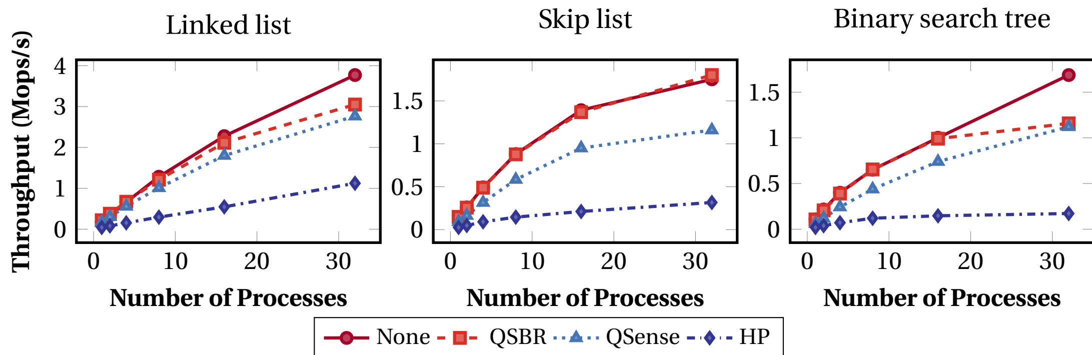


Figure 6.6 – Scalability of memory reclamation on a linked list (2000 elements), a skip list (20000 elements) and a BST (2000000 elements) with 50% updates.

system switch, every 20 seconds one of the processes is delayed for a period of 10 seconds. The operation distribution and key range sizes are fixed as above and the number of processes is fixed at 8. As before, the data structure is filled up to half of its capacity, processes start simultaneously, then run for 100 seconds. The throughput is recorded as a function of time and is measured every second.

6.7.3 Results

Figure 6.6 shows the behavior of QSBR, QSense, HP and None (the leaky implementation) on the linked list, on the skip list and on the binary search tree in the common case, when no process delays occur. The throughput of the algorithms is plotted as a function of number of cores (higher is better). Due to its amortized overhead, QSBR maintains a 2.3% overhead on average compared to None. As expected, HP achieves the lowest overall throughput, with an average overhead of 80% over the leaky implementation. This is due to the expensive per-node memory barriers used upon data structure traversal. QSense achieves two to three times better throughput than HP and has an average overhead of 29% over None. While close to QSBR in all scenarios, QSense does not match its performance. Even if QSense follows a quiescence-based reclamation scheme in the base case, it still has to maintain the hazard pointer and timestamp values updated. This induces increased overhead, compared to QSBR. Despite the fact that no memory barriers are used upon updating the hazard pointers, the process-local variable updates alone add sequential complexity. This explains why the performance gap between QSBR and QSense is larger for the skip list than for the linked list, or the tree: whereas the linked list only uses two hazard pointers per process and the tree uses six, the skip list can use up to 35 hazard pointers per process.

QSense does not completely match the performance of QSBR but, unlike QSBR, it can recover from long process delays. Figure 6.7 illustrates this. The throughput of QSense is shown alongside QSBR and HP. In this experiment, one of the processes is delayed in the 10–20, 30–40, 50–60, 70–80 and 90–100 time intervals. A similar pattern can be observed for all three

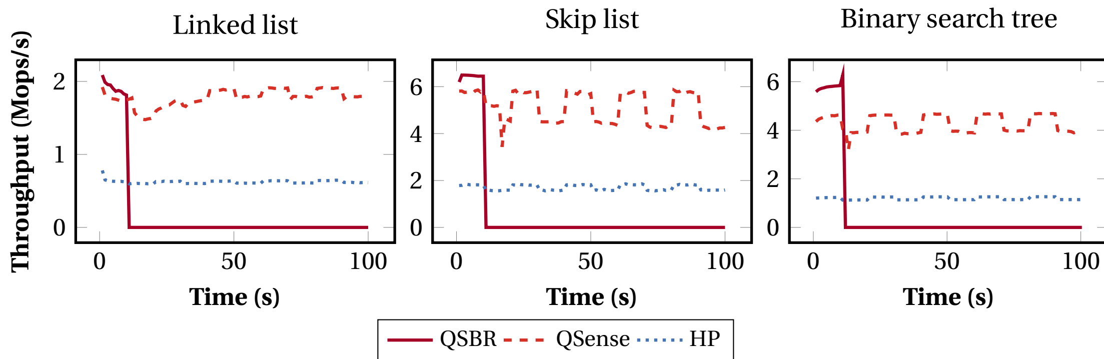


Figure 6.7 – Path switching with process delays (8 processes, 50% updates).

data structures. Note that the first time a process is delayed (after 10 seconds), QSBR can no longer quiesce. Consequently, the system runs out of memory and eventually fails. In contrast, QSense continues to run. When quiescence is no longer possible, QSense switches to Cadence and then switches back to QSBR as soon as the delayed process becomes active. The sequence of fallbacks and recoveries is continued throughout the entire experiment run. As intended, the throughput achieved by QSense is similar to QSBR during the fast path. When QSense is running in the fallback path, it can be seen that Cadence outperforms the original hazard pointer scheme by a factor of 3x on average, because it avoids per-node memory barriers upon data structure traversal.

6.8 Related Work

Reference counting (RC) [77, 94, 111, 222] assigns shared counters to each node in the data structure, representing the number of references to the nodes held at every given time. When a node's counter reaches zero, the node is safe to reclaim. Though easy to implement, RC requires expensive atomic operations on every access to maintain consistent counters.

Pointer-based techniques. Hazard pointers (HP) [179], or Pass-the-Buck [112] rely on the programmer marking nodes that are unsafe to reclaim before performing memory accesses to their locations. A reclaiming process must ensure that a node it is about to deallocate is not marked. An advantage of these schemes is that they maintain the lock-free property of data structures. Yet, their implementation requires a memory fence instruction to be issued for every node traversed, which significantly hinders the performance of read-only operations [107].

Improvements on HP. Morrison et al. [183] introduce a new, strengthened version of the Total Store Ordering (TSO) memory model [191] in which there is a known bound on the time it takes for writes in the store buffer to become visible in main memory. A variant of HP which does not need memory barriers is proposed. While this solution is similar to Cadence, it relies on hardware guarantees that do not exist yet in practice. McKenney et al. [172] describe a

method to force quiescent states by creating a high-priority daemon process that executes on each CPU in turn, but this method has never been applied to HP. Finally, Aghazadeh et al. [4] provide an improvement on HP by reducing the number of hazard-pointer-to-node comparisons per scan call to one, at the cost of increasing the amount of time between node removal and node reclamation.

Epoch-based techniques. These techniques [104, 107, 174] rely on the assumption that live processes will eventually drop references they hold on node (i.e., if a reclaiming process waits long enough, all other processes will eventually stop holding references to the deleted nodes, which will thus become safe to reclaim). Though epoch-based techniques have good performance [107], their main drawback is that they are blocking.

Ad-hoc techniques. Drop the Anchor (DTA) [40] combines timestamping with a HP-inspired technique. Processes use timestamps to track their progress and they place anchor pointers in the data structure, upon traversal. When a process delay occurs , the other processes work together to reconstruct the data structure using the anchors. Similarly to QSense, DTA spreads the cost of expensive operations across a large number of operations. However, the applicability of DTA to other data structures besides the linked list implementation provided by the authors, is unclear. In contrast, QSense is as easy to apply as HP, for which a well established methodology exists.

DEBRA+ [43] uses a variant of QSBR when processes are making progress in the common case and has a slower mechanism for treating delays. When a delayed process is preventing other processes from quiescing for too long, the slow process is *neutralized*, using an OS signal. Upon resuming execution, a neutralized process runs special *recovery code* to clean up any inconsistencies it might have left in the data structure before it was neutralized. Like QSense, DEBRA+ has a fast path and a recovery path. Unlike QSense, DEBRA+ relies on OS-specific instructions. Moreover, DEBRA+ is only easy to apply to lock-free data structures that have (1) an explicit help function and an explicit descriptor record containing all the information required by the help function, (2) operation code that follows a certain pattern (quiescent preamble – non-quiescent body – quiescent postamble) and (3) a way to perform clean-up if a process is interrupted half-way through an operation. It is unclear how to extend DEBRA+ to most state-of-the-art algorithms not satisfying the above properties.

Automatic memory reclamation. ThreadScan [8] is an automatic technique for concurrent memory reclamation. Processes add references to removed nodes to a shared *delete buffer*. Periodically, a scan is initiated by sending a signal to all processes. Each process examines its stack and registers and marks the corresponding entry in the delete buffer if there is a match, to indicate that the node is in use. After all processes complete the scan, unmarked nodes can be freed. ThreadScan amortizes the overhead of memory reclamation, similarly to QSense. Unlike QSense, ThreadScan has the advantage of being completely automatic. However, ThreadScan makes critical assumptions about the synchrony of its *worker processes* and about the layout of process stacks in memory. Cohen and Petrank [66] present an automatic memory reclamation

scheme for lock-free data structures, inspired by mark-and-sweep garbage collection. This technique relies on the data structure operations being in normalized form [221]. While such implementations of lock-free data structures exist, there is no clear methodology on how to normalize lock-free data structure implementations.

HTM techniques. Another direction for concurrent memory reclamation is the use of Hardware Transactional Memory (HTM) [114]. Dragojevic et al. [83] use HTM to produce faster and simpler solutions to a common subproblem in prior memory reclamation techniques. StackTrack [7] uses the bookkeeping facilities of HTM directly to track node references. The downside of this class of solutions is that they rely on the presence of HTM on the target machine, which is rare in practice.

6.9 Conclusion

QSense offers a fast, robust and highly applicable solution to the concurrent memory reclamation problem. Whenever possible, the fast QSBR technique is used. In case of prolonged process delays, QSense switches to Cadence, a novel hazard pointer inspired scheme, that is robust to process delays. QSense can be integrated in data structures making use of the popular hazard pointer based schemes almost effortlessly, a significant advantage from the applicability standpoint. We show experimentally that QSense achieves performance similar to the high-performing techniques in the common case (i.e., when all worker processes are active in the system), while tolerating process delays.

Concluding Remarks Part IV

7 Conclusions and Future Work

We conclude this thesis by summarizing its main findings and their implications, as well as by outlining promising directions for future work opened by this work.

7.1 Summary and Implications

At a high level, we believe this thesis takes a step toward a better understanding of the challenges and opportunities of distributed computing with modern shared memory. It is our hope that programmers can use the insights and tools in this thesis to "tame" the complexity of shared memory, be it by effectively leveraging new technologies or by adapting to changing requirements. Firstly, we have demonstrated, both in theory and in practice, the potential of RDMA to power fundamentally faster and more robust replication protocols, opening the way to replicated microsecond-level applications. Secondly, through our lower bound on the number of persistent fences, we have provided a useful point of reference for programmers aiming to use persistent memory efficiently. Furthermore, through our MCAS constructions, we have provided powerful tools to simplify the task of designing fast concurrent data structures for volatile and persistent memory. Finally, we have shown how to design a fast, robust, and widely applicable concurrent memory reclamation scheme; such schemes are becoming indispensable given the current pace of data growth.

In Part I, we focused on RDMA. We showed that RDMA's flexible and dynamic permissions enable algorithms which fundamentally improve on the traditional tradeoffs in distributed computing between performance and failure-resilence. In Chapter 2, we provided consensus protocols for the crash-fault and Byzantine-fault models with better complexity and failure-resilience than either shared memory or message passing, taken alone. Furthermore, in Chapter 3 we put these ideas to work through Mu, our RDMA-based state-machine replication system for microsecond applications.

In Part II, we considered the challenges raised by persistent memory. In Chapter 4 we showed that one persistent fence per operation is necessary and sufficient to implement any con-

current persistent object. In Chapter 5, we provided novel multi-word compare-and-swap algorithms for persistent and volatile memory. Our algorithms use significantly fewer CAS instructions per call to MCAS when compared to the state-of-the-art and, in the persistent version, much fewer persistent fences per MCAS call. Furthermore, we showed that our algorithms have near-optimal CAS usage, by providing a lower bound on the number of CASes required to implement MCAS.

In Part III, we looked at memory reclamation. We introduced QSense, a novel hybrid reclamation scheme which is fast in the common case, robust to arbitrary process delays and easy to integrate with a wide range of concurrent data structure algorithms. As part of Qsense, we described Cadence, a novel memory reclamation scheme, inspired by Hazard Pointers, but which eliminates their reliance on expensive memory barriers on the common path.

7.2 Future Directions

Practical BFT using RDMA. In Chapter 2, we show that RDMA makes it possible to solve BFT with $2f+1$ replicas such that each decision takes only 2 delays in the common case. However, our construction is not well suited for a practical implementation, due to the need for replicas to send their entire histories with every message. Thus, an interesting direction for future research is designing, implementing, and evaluating a practical RDMA-based BFT system which maintains the same high fault tolerance and low communication complexity as our theoretical construction.

Applying dynamic permissions to other problems. In Chapter 2, we demonstrate that permissioned M&M models enable consensus protocols with better fault tolerance and communication complexity than either the shared memory model or the message passing model, taken separately. This opens the question of whether RDMA permissions can enable better solutions to other distributed computing problems, such as reliable broadcast [32] (and its Byzantine variant [38]), causal-order broadcast [32], and atomic commit [97, 215].

Remote-friendly memory reclamation. Memory reclamation is especially important for RDMA-based algorithms, as their high performance makes them prone to high memory usage. It is not clear whether existing techniques for concurrent memory reclamation can be easily adapted to the RDMA case, since the latter involves the local and remote sides to synchronize on whether a given memory location can safely be reused. Furthermore, RDMA-based algorithms often require the remote side to remain passive, in order to keep latency low. This case seems even more challenging, as memory reclamation will have to be coordinated entirely from the remote side(s).

Tight bounds on MCAS complexity. Our MCAS algorithms in Chapter 5 are nearly optimal. The volatile MCAS variant requires $k+1$ CASes per call to k-CAS in the common case, whereas our lower bound states that at least k CASes are needed. The persistent MCAS variant uses the same number of CASes and 2 persistent fences, whereas our lower bound in Chapter 4 shows that at least 1 persistent fence per operation is needed to implement any concurrent object. It remains an open problem to design volatile and persistent MCAS algorithms that match these lower bounds, or prove tighter lower bounds that match our proposed algorithms.

Appendices Part V

A The Impact of RDMA on Agreement

A.1 Correctness of Reliable Broadcast

Observation A.1.1. *In Algorithm 2.1, if p is a correct process, then no slot that belongs to p is written to more than once.*

Proof. Since p is correct, p never writes on any slot more than once. Furthermore, since all slots are single-writer registers, no other process can write on these slots. □

Observation A.1.2. *In Algorithm 2.1, correct processes invoke and return from try_deliver(q) infinitely often, for all $q \in \Pi$.*

Proof. The try_deliver() function does not contain any blocking steps, loops or goto statements. Thus, if a correct process invokes try_deliver(), it will eventually return. Therefore, for a fixed q the infinite loop at line 15 will invoke and return try_deliver(q) infinitely often. Since the parallel for loop at line 14 performs the infinite loop in parallel for each $q \in \Pi$, the Observation holds. □

Proof of Lemma 2.4.2. We prove the lemma by showing that Algorithm 2.1 correctly implements reliable broadcast. That is, we need to show that Algorithm 2.1 satisfies the four properties of reliable broadcast.

Property 1. Let p be a correct process that broadcasts (k, m). We show that all correct processes eventually deliver (k, m) from p. Assume by contradiction that there exists some correct process q which does not deliver (k, m). Furthermore, assume without loss of generality that k is the smallest key for which Property 1 is broken. That is, all correct processes must eventually deliver all messages (k', m') from p, for $k' < k$. Thus, all correct processes must eventually increment $last[p]$ to k.

We consider two cases, depending on whether or not some process eventually writes an L2 proof for some (k, m') message from p in its L2Proof slot.

First consider the case where no process ever writes an L2 proof of any value (k, m') from p. Since p is correct, upon broadcasting (k, m), p must sign and write (k, m) into Value[p,k,p] at line 12. By Observation A.1.1, (k, m) will remain in that slot forever. Because of this, and because there is no L2 proof, all correct processes, after reaching $last[p] = k$, will eventually read (k, m) in line 26, write it into their own Value slot in line 28 and change their state to WaitForL1Proof.

Furthermore, since p is correct and we assume signatures are unforgeable, no process q can write any other valid value $(k', m') \neq (k, m)$ into its $Values[q, k, p]$ slot. Thus, eventually each correct process q will add at least $f + 1$ copies of (k, m) to its checkedVals, write an L1proof consisting of these values into L1Proof[q,k,p] in line 37, and change their state to WaitForL2Proof.

Therefore, all correct processes will eventually read at least $f + 1$ valid L1 Proofs for (k, m) in line 42 and construct and write valid L2 proofs for (k, m). This contradicts the assumption that no L2 proof ever gets written.

In the case where there is some L2 proof, by the argument above, the only value it can prove is (k, m). Therefore, all correct processes will see at least one valid L2 proof at deliver. This contradicts our assumption that q is correct but does not deliver (k, m) from p.

Property 2. We now prove the second property of reliable broadcast. Let p and p' be any two correct processes, and q be some process, such that p delivers (k, m) from q and p' delivers (k, m') from q. Assume by contradiction that $m \neq m'$.

Since p and p' are correct, they must have seen valid L2 proofs at line 50 before delivering (k, m) and (k, m') respectively. Let $\mathscr{P}$ and $\mathscr{P}'$ be those valid proofs for (k, m) and (k, m') respectively. $\mathscr{P}$ (resp. $\mathscr{P}'$) consists of at least $f + 1$ valid L1 proofs; therefore, at least one of those proofs was created by some correct process r (resp. r'). Since r (resp. r') is correct, it must have written (k, m) (resp. (k, m')) to its Values slot in line 28. Note that after copying a value to their slot, in the WaitForL1Proof state, correct processes read *all* Value slots line 33. Thus, both r and r' read all Value slots before compiling their L1 proof for (k, m) (resp. (k, m')).

Assume without loss of generality that r wrote (k, m) before r' wrote (k, m'); by Observation A.1.1, it must then be the case that r' later saw both (k, m) and (k, m') when it read all Values slots (line 33). Since r' is correct, it cannot have then compiled an L1 proof for (k, m') (the check at line 35 failed). We have reached a contradiction.

Property 3. We show that if a correct process p delivers (k, m) from a correct process p', then p' broadcast (k, m). Correct processes only deliver values for which a valid L2 proof exists (lines 50—21). Therefore, p must have seen a valid L2 proof $\mathscr{P}$ for (k, m). $\mathscr{P}$ consists of at least $f + 1$ L1 proofs for (k, m) and each L1 proof consists of at least $f + 1$ matching copies of (k, m), signed by p'. Since p' is correct and we assume signatures are unforgeable, p' must have broadcast (k, m) (otherwise p' would not have attached its signature to (k, m)).

Property 4. Let p be a correct process such that p delivers (k,m) from q. We show that all correct process must deliver (k,m') from q, for some m'.

By construction of the algorithm, if p delivers (k,m) from q, then for all $i<k$ there exists m_i such that p delivered (i,m_i) from q before delivering (k,m) (this is because p can only deliver (k,m) if $last[q]=k$ and $last[q]$ is only incremented to k after p delivers $(k-1,m_{k-1})$).

Assume by contradiction that there exists some correct process r which does not deliver (k,m') from q, for any m'. Further assume without loss of generality that k is the smallest key for which r does not deliver any message from q. Thus, r must have delivered (i,m'_i) from q for all $i<k$; thus, r must have incremented $last[q]$ to k. Since r never delivers any message from q for key k, r's $last[q]$ will never increase past k.

Since p delivers (k,m) from q, then p must have written a valid L2 proof $\mathscr{P}$ of (k,m) in its L2Proof slot in line 52. By Observation A.1.1, $\mathscr{P}$ will remain in p's L2Proof[p,k,q] slot forever. Thus, at least one of the slots $L2Proof[\cdot,k,q]$ will forever contain a valid L2 proof. Since r's $last[q]$ eventually reaches k and never increases past k, r will eventually (by Observation A.1.2) see a valid L2 proof in line 50 and deliver a message for key k from q. We have reached a contradiction. □

A.2 Correctness of Cheap Quorum

We prove that Cheap Quorum satisfies certain useful properties that will help us show that it composes with Preferential Paxos to form a correct weak Byzantine agreement protocol. For the proofs, we first formalize some terminology. We say that a process *proposed* a value v by time t if it successfully executes line 4; that is, p receives the response ack in line 4 by t. When a process aborts, note that it outputs a tuple. We say that the first element of its tuple is its *abort value*, and the second is its *abort proof.* We sometimes say that a process p aborts with value v and proof pr, meaning that p outputs (v,pr) in its abort. Furthermore, the value in a process p's Proof region is called a *correct unanimity proof* if it contains n copies of the same value, each correctly signed by a different process.

Observation A.2.1. *In Cheap Quorum, no value written by a correct process is ever overwritten.*

Proof. By inspecting the code, we can see that the correct behavior is for processes to never overwrite any values. Furthermore, since all regions are initially single-writer, and the legalChange function never allows another process to acquire write permission on a region that they cannot write to initially, no other process can overwrite these values. □

Lemma A.2.2 (Cheap Quorum Validity)**.** *In Cheap Quorum, if there are no faulty processes and some process decides v, then v is the input of some process.*

Proof. If $p=p_1$, the lemma is trivially true, because p_1 can only decide on its input value. If

$p \neq p_1$, p can only decide on a value v if it read that value from the leader's region. Since only the leader can write to its region, it follows that p can only decide on a value that was proposed by the leader (p_1). □

Lemma A.2.3 (Cheap Quorum Termination)**.** *If a correct process p proposes some value, every correct process q will decide a value or abort.*

Proof. Clearly, if $q = p_1$ proposes a value, then q decides. Now let $q \neq p_1$ be a correct follower and assume p_1 is a correct leader that proposes v. Since p_1 proposed v, p_1 was able to write v in the leader region, where v remains forever by Observation A.2.1. Clearly, if q eventually enters panic mode, then it eventually aborts; there is no waiting done in panic mode. If q never enters panic mode, then q eventually sees v on the leader region and eventually finds $2f + 1$ copies of v on the regions of other followers (otherwise q would enter panic mode). Thus q eventually decides v. □

Lemma A.2.4 (Cheap Quorum Progress)**.** *If the system is synchronous and all processes are correct, then no correct process aborts in Cheap Quorum.*

Proof. Assume the contrary: there exists an execution in which the system is synchronous and all processes are correct, yet some process aborts. Processes can only abort after entering panic mode, so let t be the first time when a process enters panic mode and let p be that process. Since p cannot have seen any other process declare panic, p must have either timed out at line 12 or 22, or its checks failed on line 13. However, since the entire system is synchronous and p is correct, p could not have panicked because of a time-out at line 12. So, p_1 must have written its value v, correctly signed, to p_1's region at a time $t' < t$. Therefore, p also could not have panicked by failing its checks on line 13. Finally, since all processes are correct and the system is synchronous, all processes must have seen p_1's value and copied it to their slot. Thus, p must have seen these values and decided on v at line 20, contradicting the assumption that p entered panic mode. □

Lemma A.2.5 (Lemma 2.4.7: Cheap Quorum Decision Agreement)**.** *Let p and q be correct processes. If p decides v_1 while q decides v_2, then $v_1 = v_2$.*

Proof. Assume the property does not hold: p decided some value v_1 and q decided some different value v_2. Since p decided v_1, then p must have seen a copy of v_1 at $2f_P + 1$ replicas, including q. But then q cannot have decided v_2, because by Observation A.2.1, v_1 never gets overwritten from q's region, and by the code, q only can decide a value written in its region. □

Lemma A.2.6 (Lemma 2.4.8: Cheap Quorum Abort Agreement)**.** *Let p and q be correct processes (possibly identical). If p decides v in Cheap Quorum while q aborts from Cheap Quorum, then v will be q's abort value. Furthermore, if p is a follower, q's abort proof is a correct unanimity proof.*

Proof. If $p = q$, the property follows immediately, because of lines 4 through 6 of panic mode. If $p \neq q$, we consider two cases:

- If p is a follower, then for p to decide, all processes, and in particular, q, must have replicated both v and a correct proof of unanimity before p decided. Therefore, by Observation A.2.1, v and the unanimity proof are still there when q executes the panic code in lines 4 through 6. Therefore q will abort with v as its value and a correct unanimity proof as its abort proof.
- If p is the leader, then first note that since p is correct, by Observation A.2.1 v remains the value written in the leader's Value region. There are two cases. If q has replicated a value into its Value region, then it must have read it from $Value[p_1]$, and therefore it must be v. Again by Observation A.2.1, v must still be the value written in q's Value region when q executes the panic code. Therefore q aborts with value v. Otherwise, if q has not replicated a value, then q's Value region must be empty at the time of the panic, since the legalChange function disallows other processes from writing on that region. Therefore q reads v from $Value[p_1]$ and aborts with v. □

Lemma A.2.7. *Cheap Quorum is 2-deciding.*

Proof. Consider an execution in which every process is correct and the system is synchronous. Then no process will enter panic mode (by Lemma A.2.4) and thus p_1 will not have its permission revoked. p_1 will therefore be able to write its input value to p_1's region and decide after this single write (2 delays). □

A.3 Correctness of the Fast & Robust

We now prove the following key composition property that shows that the composition of Cheap Quorum and Preferential Paxos is safe.

Lemma A.3.1 (Lemma 2.4.11: Composition Lemma)**.** *If some correct process decides a value v in Cheap Quorum before an abort, then v is the only value that can be decided in Preferential Paxos with priorities as defined in Definition 2.4.10.*

Proof. To prove this lemma, we mainly rely on two properties: the Cheap Quorum Abort Agreement (Lemma 2.4.8) and Preferential Paxos Priority Decision (Lemma 2.4.9). We consider two cases.

Case 1. Some correct follower process $p \neq p_1$ decided v in Cheap Quorum. Then note that by Lemma 2.4.8, all correct processes aborted with value v and a correct unanimity proof. Since $n \geq 2f + 1$, there are at least $f + 1$ correct processes. Note that by the way we assign priorities to inputs of Preferential Paxos in the composition of the two algorithms, all correct processes have inputs with the highest priority. Therefore, by Lemma 2.4.9, the only decision

value possible in Preferential Paxos is v. Furthermore, note that by Lemma 2.4.7, if any other correct process decided in Cheap Quorum, that process's decision value was also v.

Case 2. Only the leader, p_1, decides in Cheap Quorum, and p_1 is correct. Then by Lemma 2.4.8, all correct processes aborted with value v. Since p_1 is correct, v is signed by p_1. It is possible that some of the processes also had a correct unanimity proof as their abort proof. However, note that in this scenario, all correct processes (at least $f+1$ processes) had inputs with either the highest or second highest priorities, all with the same abort value. Therefore, by Lemma 2.4.9, the decision value must have been the value of one of these inputs. Since all these inputs had the same value v, v must be the decision value of Preferential Paxos. □

Theorem A.3.2 (End-to-end Validity)**.** *In the Fast & Robust algorithm, if there are no faulty processes and some process decides v, then v is the input of some process.*

Proof. Note that by Lemmas A.2.2 and 2.4.9, this holds for each of the algorithms individually. Furthermore, recall that the abort values of Cheap Quorum become the input values of Preferential Paxos, and the set-up phase does not invent new values. Therefore, we just have to show that if Cheap Quorum aborts, then all abort values are inputs of some process. Note that by the code in panic mode, if Cheap Quorum aborts, a process p can output an abort value from one of three sources: its own Value region, the leader's Value region, or its own input value. Clearly, if its abort value is its input, then we are done. Furthermore note that a correct leader only writes its input in the Value region, and correct followers only write a copy of the leader's Value region in their own region. Since there are no faults, this means that only the input of the leader may be written in any Value region, and therefore all processes always abort with some processes input as their abort value. □

Theorem A.3.3 (End-to-end Agreement)**.** *In the Fast & Robust algorithm, if p and q are correct processes such that p decides v_1 and q decides v_2, then $v_1 = v_2$.*

Proof. First note that by Lemmas 2.4.7 and 2.4.9, each of the algorithms satisfy this individually. Thus Lemma 2.4.11 implies the theorem. □

Theorem A.3.4 (End-to-end Termination)**.** *In Fast & Robust algorithm, if some correct process is eventually the sole leader forever, then every correct process eventually decides.*

Proof. Assume towards a contradiction that some correct process p is eventually the sole leader forever, and let t be the time when p last becomes leader. Now consider some process q that has not decided before t. We consider several cases:

1. If q is executing Preferential Paxos at time t, then q will eventually decide, by termination of Preferential Paxos (Lemma 2.4.9).

2. If q is executing Cheap Quorum at time t, we distinguish two sub-cases:

(a) p is also executing as the leader of Cheap Quorum at time t. Then p will eventually propose a value, so q will either decide in Cheap Quorum or abort from Cheap Quorum (by Lemma A.2.3) and decide in Preferential Paxos by Lemma 2.4.9.

(b) p is executing in Preferential Paxos. Then p must have panicked and aborted from Cheap Quorum. Thus, q will also abort from Cheap Quorum and decide in Preferential Paxos by Lemma 2.4.9. □

Note that to strengthen A.3.4 to general termination as stated in our model, we require the additional standard assumption [146] that some correct process p is eventually the sole leader forever. In practice, however, p does not need to be the sole leader forever, but rather *long enough* so that all correct processes decide.

A.4 Correctness of Protected Memory Paxos

In this section, we present the proof of Theorem 2.5.1. We do so by showing the Algorithm 2.6 is an algorithm that satisfies all of the properties in the theorem.

We first show that Algorithm 2.6 correctly implements consensus, starting with validity. Intuitively, validity is preserved because each process that writes any value in a slot either writes its own value, or adopts a value that was previously written in a slot. We show that every value written in any slot must have been the input of some process.

Theorem A.4.1 (Validity)**.** *In Algorithm 2.6, if a process p decides a value v, then v was the input to some process.*

Proof. Assume by contradiction that some process p decides a value v and v is not the input of any process. Since v is not the input value of p, then p must have adopted v by reading it from some process p' at line 23. Note also that a process cannot adopt the initial value $\perp$, and thus, v must have been written in p''s memory by some other process. Thus we can define a sequence of processes $s_1, s_2, \ldots, s_k$, where s_i adopts v from the location where it was written by s_{i+1} and $s_1 = p$. This sequence is necessarily finite since there have been a finite number of steps taken up to the point when p decided v. Therefore, there must be a last element of the sequence, s_k who wrote v in line 35 without having adopted v. This implies v was s_k's input value, a contradiction. □

We now focus on agreement.

Theorem A.4.2 (Agreement)**.** *In Algorithm 2.6, for any processes p and q, if p and q decide values v_p and v_q respectively, then $v_p = v_q$.*

Before showing the proof of the theorem, we first introduce the following useful lemmas.

Lemma A.4.3. *The values a leader accesses on remote memory cannot change between when it reads them and when it writes them.*

Proof. Recall that each memory only allows write-access to the most recent process that acquired it. In particular, that means that each memory only gives access to one process at a time. Note that the only place at which a process acquires write-permissions on a memory is at the very beginning of its run, before reading the values written on the memory. In particular, for each memory d a process does not issue a read on d before its permission request on d successfully completes. Therefore, if a process p succeeds in writing on memory m, then no other process could have acquired d's write-permission after p did, and therefore, no other process could have changed the values written on m after p's read of m. □

Lemma A.4.4. *If a leader writes values v_i and v_j at line 35 with the same proposal number to memories i and j, respectively, then $v_i = v_j$.*

Proof. Assume by contradiction that a leader p writes different values $v_1 \neq v_2$ with the same proposal number. Since each thread of p executes the phase 2 write (line 35) at most once per proposal number, it must be that different threads T_1 and T_2 of p wrote v_1 and v_2, respectively. If p does not perform phase 1 (i.e., if $p = p_1$ and this is p's first attempt), then it is impossible for T_1 and T_2 to write different values, since CurrentVal was set to v at line 14 and was not changed afterwards. Otherwise (if p performs phase 1), then let t_1 and t_2 be the times when T_1 and T_2 executed line 8, respectively (T_1 and T_2 must have done so, since we assume that they both reached the phase 2 write at line 35). Assume *wlog* that $t_1 \leq t_2$. Due to the check and abort at line 29, CurrentVal cannot change after t_1 while keeping the same proposal number. Thus, when T_1 and T_2 perform their phase 2 writes (after t_1), CurrentVal has the same value as it did at t_1; it is therefore impossible for T_1 and T_2 to write different values. We have reached a contradiction. □

Lemma A.4.5. *If a process p performs phase 1 and then writes to some memory m with proposal number b at line 35, then p must have written b to m at line 21 and read from m at line 23.*

Proof. Let T be the thread of p which writes to m at line 35. If phase 1 is performed (i.e., the condition at line 19 is satisfied), then by construction T cannot reach line 35 without first performing lines 21 and 23. Since T only communicates with m, it must be that lines 21 and 23 are performed on m. □

Proof of Theorem A.4.2. Assume by contradiction that $v_p \neq v_q$. Let b_p and b_q be the proposal numbers with which v_p and v_q are decided, respectively. Let W_p (resp. W_q) be the set of memories to which p (resp. q) successfully wrote in phase 2 line 35 before deciding v_p (resp. v_q). Since W_p and W_q are both majorities, their intersection must be non-empty. Let m be any memory in $W_p \cap W_q$.

We first consider the case in which one of the processes did not perform phase 1 before deciding (i.e., one of the processes is p_1 and it decided on its first attempt). Let that process be p *wlog*. Further assume *wlog* that q is the first process to enter phase 2 with a value different from v_p. p's phase 2 write on m must have preceded q obtaining permissions from m (otherwise, p's write would have failed due to lack of permissions). Thus, q must have seen p's value during its read on m at line 23, and thus q cannot have adopted its own value. Since q is the first process to enter phase 2 with a value different from v_p, q cannot have adopted any other value than v_p, so q must have adopted v_p. Contradiction.

We now consider the remaining case: both p and q performed phase 1 before deciding. We assume *wlog* that $b_p < b_q$ and that b_q is the smallest proposal number larger than b_p for which some process enters phase 2 with CurrentVal $\neq v_p$.

Since $b_p < b_q$, p's read at m must have preceded q's phase 1 write at m (otherwise p would have seen q's larger proposal number and aborted). This implies that p's phase 2 write at m must have preceded q's phase 1 write at m (by Lemma A.4.3). Thus q must have seen v_p during its read and cannot have adopted its own input value. However, q cannot have adopted v_p, so q must have adopted v_q from some other slot sl that q saw during its read. It must be that $sl.minProposal < b_q$, otherwise q would have aborted. Since $sl.minProposal \geq sl.accProposal$ for any slot, it follows that $sl.accProposal < b_q$. If $sl.accProposal < b_p$, q cannot have adopted $sl.value$ in line 30 (it would have adopted v_p instead). Thus it must be that $b_p \leq sl.accProposal < b_q$; however, this is impossible because we assumed that b_q is the smallest proposal number larger than b_p for which some process enters phase 2 with CurrentVal $\neq v_p$. We have reached a contradiction. □

Finally, we prove that the termination property holds.

Theorem A.4.6 (Termination)**.** *Eventually, all correct processes decide.*

Lemma A.4.7. *If a correct process p is executing the for loop at lines 18–37, then p will eventually exit from the loop.*

Proof. The threads of the for loop perform the following potentially blocking steps: obtaining permission (line 20), writing (lines 21 and 35), reading (line 23), and waiting for other threads (the barrier at line 7 and the exit condition at line 37). By our assumption that a majority of memories are correct, a majority of the threads of the for loop must eventually obtain permission in line 20 and invoke the write at line 21. If one of these writes fails due to lack of permission, the loop aborts and we are done. Otherwise, a majority of threads will perform the read at line 23. If some thread aborts at lines 22 and 26, then the loop aborts and we are done. Otherwise, a majority of threads must add themselves to ListOfReady, pass the barrier at line 7 and invoke the write at line 35. If some such write fails, the loop aborts; otherwise, a majority of threads will reach the check at line 37 and thus the loop will exit. □

Proof of Theorem A.4.6. The Ω failure detector guarantees that eventually, all processes trust the same correct process p. Let t be the time after which all correct processes trust p forever. By Lemma A.4.7, at some time $t' \geq t$, all correct processes except p will be blocked at line 12. Therefore, the *minProposal* values of all memories, on all slots except those of p stop increasing. Thus, eventually, p picks a *propNr* that is larger than all others written on any memory, and stops restarting at line 26. Furthermore, since no process other than p is executing any steps of the algorithm, and in particular, no process other than p ever acquires any memory after time t', p never loses its permission on any of the memories. So, all writes executed by p on any correct memory must return ack. Therefore, p will decide and broadcast its decision to all. All correct processes will receive p's decision and decide as well. □

To complete the proof of Theorem 2.5.1, we now show that Algorithm 2.6 is 2-deciding.

Theorem A.4.8. *Algorithm 2.6 is 2-deciding.*

Proof. Consider an execution in which p_1 is timely, and no process's failure detector ever suspects p_1. Then, since no process thinks itself the leader, and processes do not deviate from their protocols, no process calls changePermission on any memory. Furthermore, p_1's does not perform phase 1 (lines 19–33), since it is p_1's first attempt. Thus, since p_1 initially has write permission on all memories, all of p_1's phase 2 writes succeed. Therefore, p_1 terminates, deciding its own proposed value v, after one write per memory. □

B Microsecond Consensus for Microsecond Applications

B.1 Pseudocode of the Basic Version

```
1  Propose(myValue):
2      done = false
3      If I just became leader or I just aborted
4          For every process p in parallel:
5              Request permission from p
6              If p acks, add p to confirmedFollowers
7          Until this has been done for a majority of processes
8      While not done:
9          Execute Prepare Phase
10         Execute Accept Phase
```

```
11 struct Log {
12     log[] = ⊥ for all slots
13     minProposal = 0
14     FUO = 0    }

16 Prepare Phase:
17     Pick a new proposal number, propNum, that is higher than any seen so
         ↪ far
18     For every process p in confirmedFollowers:
19         Read minProposal from p's log
20         Abort if any read fails
21     If propNum < some minProposal read, abort
22     For every process p in confirmedFollowers:
23         Write propNum into LOG[p].minProposal
24         Read LOG[p].slots[myFUO]
25         Abort if any write or read fails
26     if all entries read were empty:
27         value = myValue
28     else:
29         value = entry value with the largest proposal number of slots
             ↪ read
```

```
30 Accept Phase:
31     For every process p in confirmedFollowers:
32         Write value,propNum to p in slot myFUO
33         Abort if any write fails
34     If value == myValue:
35         done = true
36     Locally increment myFUO
```

Note that write permission can only be granted at most once per request; it is impossible to send a single permission request, be granted permission, lose permission and then regain it without issuing a new permission request. This is the way that permission requests work in our implementation, and is key for the correctness argument to go through; in particular, it is important that a leader cannot lose permission between two of its writes to the same follower without being aware that it lost permission.

B.2 Definitions

Definition B.2.1 (Quorum)**.** *A quorum is any set that contains at least a majority of the processes.*

Definition B.2.2 (Decided Value)**.** *We say that a value v is decided at index i if there exists a quorum Q such that for every process $p \in Q$, p's log contains v at index i.*

Definition B.2.3 (Committed Value)**.** *We say that a value v is committed at process p at index i if p's log contains v at index i, such that i is less than p's FUO.*

B.3 Invariants

Invariant B.3.1 (Committed implies decided)**.** *If a value v is committed at some process p at index i, then v is decided at index i.*

Proof. Assume v is committed at some process p at index i. Then p must have incremented its FUO past i at line 36, therefore p must have written v at a majority at line 32. □

Invariant B.3.2 (Values are never erased)**.** *If a log slot contains a value at time t, that log slot will always contain some value after time t.*

Proof. By construction of the algorithm, value are never erased (note: values can be overwritten, but only with a non-$\perp$ value). □

B.3.1 Validity

Invariant B.3.3. *If a log slot contains a value $v \neq \perp$, then v is the input of some process.*

Proof. Assume the contrary and let t be the earliest time when some log slot (call it L) contained a non-input value (call it v). In order for L to contain v, some process p must have written v into L at line 32. Thus, either v was the input value of p (which would lead to a contradiction), or p adopted v at line 29, after reading it from some log slot L' at line 24. Thus, L' must have contained v earlier than t, a contradiction of our choice of t. □

Theorem B.3.4 (Validity)**.** *If a value v is committed at some process, then v was the input value of some process.*

Proof. Follows immediately from Invariant B.3.3 and the definition of being committed. □

B.3.2 Agreement

Invariant B.3.5 (Solo detection)**.** *If a process p writes to a process q in line 23 or in line 32, then no other process r wrote to q since p added q to its confirmed followers set.*

Proof. Assume the contrary: p added q to its confirmed followers set at time t_0 and wrote to q at time $t_2 > t_0$; $r \neq p$ wrote to q at time t_1, $t_0 < t_1 < t_2$. Then:

1. r had write permission on q at t_1.
2. p had write permission on q at t_2.
3. (From (1) and (2)) p must have obtained write permission on q between t_1 and t_2. But this is impossible, since p added q to its confirmed followers set at $t_0 < t_1$ and thus p must have obtained permission on q before t_0. By the algorithm, p did not request permission on q again since obtaining it, and by the way permission requests work, permission is granted at most once per request. We have reached a contradiction. □

Invariant B.3.6. *If some process p_1 successfully writes value v_1 and proposal number b_1 to its confirmed followers in slot i at line 32, then any process p_2 entering the accept phase with proposal number $b_2 > b_1$ for slot i will do so with value v_1.*

Proof. Assume the contrary: some process enters the accept phase for slot i with a proposal number larger than b_1, with a value $v_2 \neq v_1$. Let p_2 be the first such process to enter the accept phase.

Let C_1 (resp. C_2) be the confirmed followers set of p_1 (resp. p_2). Since C_1 and C_2 are both quorums, they must intersect in at least one process, call it q. Since q is in the confirmed followers set of both p_1 and p_2, both must have read its minProposal (line 19), written its minProposal with their own proposal value (line 23) and read its ith log slot (line 24). Furthermore, p_1 must have written its new value into that slot (line 32). Note that since p_1 successfully wrote value v_1 on q, by Invariant B.3.5, p_2 could not have written on q between the time at which p_1 obtained

its permission on it and the time of p_1's write on q's ith slot. Thus, p_2 either executed both of its writes on q before p_1 obtained permissions on q, or after p_1 wrote its value in q's ith slot. If p_2 executed its writes before p_1, then p_1 must have seen p_2's proposal number when reading q's minProposal in line 19 (since p_1 obtains permissions before executing this line). Thus, p_1 would have aborted its attempt and chosen a higher proposal number, contradicting the assumption that $b_1 < b_2$.

Thus, p_2 must have executed its first write on q after p_1 executed its write of v_1 in q's log. Since p_2's read of q's slot happens after its first write (in line 24), this read must have happened after p_1's write, and therefore p_2 saw v_1, b_1 in q's ith slot. By assumption, p_2 did not adopt v_1. By line 29, this means p_2 read v_2 with a higher proposal number than b_1 from some other process in C_2. This contradicts the assumption that p_2 was the first process to enter the accept phase with a value other than v_1 and a proposal number higher than b_1. □

Theorem B.3.7 (Agreement)**.** *If v_1 is committed at p_1 at index i and v_2 is committed at p_2 at index i, then $v_1 = v_2$.*

Proof. In order for v_1 (resp. v_2) to be committed at p_1 (resp. p_2) at index i, p_1 (resp. p_2) must have incremented its FUO past i and thus must have successfully written v_1 (resp. v_2) to its confirmed follower set at line 32. Let b_1 (resp. b_2) be the proposal number p_1 (resp. p_2) used at line 32. Assume without loss of generality that $b_1 < b_2$. Then, by Invariant B.3.6, p_2 must have entered its accept phase with value v_1 and thus must have written v_1 to its confirmed followers at line 32. Therefore, $v_1 = v_2$. □

B.3.3 Termination

Invariant B.3.8 (Termination implies commitment.)**.** *If a process p calls propose with value v and returns from the propose call, then v is committed at p.*

Proof. Follows from the algorithm: p returns from the propose call only after it sees $done$ to be $true$ at line 8; for this to happen, p must set $done$ to $true$ at line 35 and increment its FUO at line 36. In order for p to set $done$ to $true$, p must have successfully written some value val to its confirmed follower set at line 32 and val must be equal to v (check at line 34). Thus, when p increments its FUO at line 36, v becomes committed at p. □

Invariant B.3.9 (Weak Termination)**.** *If a correct process p invokes Propose and does not abort, then p eventually returns from the call.*

Proof. The algorithm does not have any blocking steps or goto statements, and has only one unbounded loop at line 8. Thus, we show that p will eventually exit the loop at line 8.

Let t be the time immediately after p finishes constructing its confirmed followers set (lines 4–7). Let i be the highest index such that one of p's confirmed followers contains a value in its

log at index i at time t. Given that p does not abort, it must be that p does not lose write permission on any of its confirmed followers and thus has write permission on a quorum for the duration of its call. Thus, after time t and until the end of p's call, no process is able to write any new value at any of p's confirmed followers $[*]$.

Since p never aborts, it will repeatedly execute the accept phase and increment its FUO at line 36 until p's FUO is larger than i. During its following prepare phase, p will find all slots to be empty (due to $[*]$) and adopt its own value v at line 27. Since p does not abort, it must be that p succeeds in writing v to its confirmed followers at line 32 and sets $done$ to $true$ in line 35. Thus, p eventually exits the loop at line 8 and returns. □

Theorem B.3.10 (Termination)**.** *If eventually there is a unique non-faulty leader, then eventually every Propose call returns.*

Proof. We show that eventually p does not abort from any Propose call and thus, by Invariant B.3.9, eventually p returns from every Propose call.

Consider a time t such that (1) no processes crash after t and (2) a unique process p considers itself leader forever after t.

Furthermore, by Invariant B.3.9, by some time $t' > t$ all correct processes will return or abort from any Propose call they started before t; no process apart from p will call Propose again after t' since p is the unique leader.

Thus, in any propose call p starts after t', p will obtain permission from a quorum in lines 4–7 and will never lose any permissions (since no other process is requesting permissions). Thus, all of p's reads and writes will succeed, so p will not abort at lines 20, 25, or 33.

Furthermore, since no process invokes Propose after t', the minProposals of p confirmed followers do not change after this time. Thus, by repeatedly increasing its minProposal at line 17, p will eventually have the highest proposal number among its confirmed followers, so p will not abort at line 21.

Therefore, by Invariant B.3.9, p will eventually return from every Propose call. □

B.4 Optimizations & Additions

B.4.1 New Leader Catch-Up

In the basic version of the algorithm described so far, it is possible for a new leader to miss decided entries from its log (e.g., if the new leader was not part of the previous leader's confirmed followers). The new leader can only catch up by attempting to propose new values at its current FUO, discovering previously accepted values, and re-committing them. This is correct but inefficient.

We describe an extension that allows a new leader ℓ to catch up faster: after constructing its confirmed followers set (lines 4–7), ℓ can read the FUO of each of its confirmed followers, determine the follower f with the highest FUO, and bring its own log and FUO up to date with f. This is described in the pseudocode below:

Algorithm B.1 – Optimization: Leader Catch Up.

```
1  For every process p in confirmedFollowers
2      Read p's FUO
3      Abort if any read fails
4  F = follower with max FUO
5  if F.FUO > my_FUO:
6      Copy F.LOG[my_FUO: F.FUO] into my log
7      myFUO = F.FUO
8      Abort if any read fails
```

We defer our correctness argument for this extension to Section B.4.2.

B.4.2 Update Followers

While the previous extension allows a new leader to catch up in case it does not have the latest committed values, followers' logs may still be left behind (e.g., for those followers that were not part of the leader's confirmed followers).

As is standard for practical Paxos implementations, we describe a mechanism for followers' logs to be updated so that they contain all committed entries that the leader is aware of. After a new leader ℓ updates its own log as described in Algorithm B.1, it also updates its confirmed followers' logs and FUOs:

Algorithm B.2 – Optimization: Update Followers.

```
1  For every process p in confirmed followers:
2      Copy myLog[p.FUO: my_FUO] into p.LOG
3      p.FUO = my_FUO
4      Abort if any write fails
```

We now argue the correctness of the update mechanisms in this and the preceding subsections. These approaches clearly do not violate termination. We now show that they preserve agreement and validity.

Validity. We extend the proof of Invariant B.3.3 to also cover Algorithms B.1 and B.2; the proof of Theorem B.3.4 remains unchanged.

Assume by contradiction that some log slot L does not satisfy Invariant B.3.3. Without loss of generality, assume that L is the first log slot in the execution which stops satisfying Invariant B.3.3. In order for L to contain v, either (i) some process q wrote v into L at line 32, or (ii) v was copied into L using Algorithm B.1 or B.2. In case (i), either v was q's input value (a

contradiction), or q adopted v at line 29 after reading it from some log slot $L' \neq L$. In this case, L' must have contained v before L did, a contradiction of our choice of L. In case (ii), some log slot L'' must have contained v before L did, again a contradiction. □

Agreement. We extend the proof of B.3.7 to also cover Algorithms B.1 and B.2. Let t be the earliest time when agreement is broken; i.e., t is the earliest time such that, by time t, some process p_1 has committed v_1 at i and some process p_2 has committed $v_2 \neq v_1$ at i. We can assume without loss of generality that p_1 commits v_1 at $t_1 = t$ and p_2 commits v_2 at $t_2 < t_1$. We now consider three cases:

1. Both p_1 and p_2 commit normally by incrementing their FUO at line 36. Then the proof of B.3.7 applies to p_1 and p_2.
2. p_1 commits normally by incrementing its FUO at line 36, while p_2 commits with Algorithm B.1 or B.2. Then some process p_3 must have committed v_2 normally at line 36 and the proof of B.3.7 applies to p_1 and p_3.
3. p_1 commits v_1 using Algorithm B.1 or B.2. Then v_1 was copied to p_1's log from some other process p_3's log, where v_1 had already been committed. But then, agreement must have been broken earlier than t (v_1 committed at p_3, v_2 committed at p_2), a contradiction.

□

B.4.3 Followers Update Their Own FUO

In the algorithm and optimizations presented so far, the only way for the FUO of a process p to be updated is by the leader; either by p being the leader and updating its own FUO, or by p being the follower of some leader that executes Algorithm B.2. However, in the steady state, when the leader doesn't change, it would be ideal for a follower to be able to update its own FUO. This is especially important in practice for SMR, where each follower should be applying committed entries to its local state machine. Thus, knowing which entries are committed as soon as possible is crucial. For this purpose, we introduce another simple optimization, whereby a follower updates its own FUO to i if it has a non-empty entry in some slot $j \geq i$ and all slots $k < i$ are populated.

Algorithm B.3 – Optimization: Followers Update Their Own FUO.

```
1 if LOG[i] ≠⊥ && my_FUO == i-1
2     my_FUO = i
```

Note that this optimization doesn't write any new values on any slot in the log, and therefore, cannot break Validity. Furthermore, since it does not introduce any waiting, it cannot break termination. We now prove that this doesn't break Agreement.

Agreement. Assume by contradiction that executing Algorithm B.3 can break agreement. Let p be the first process whose execution of Algorithm B.3 breaks agreement, and let t be the time at which it changes its FUO to i, thereby breaking agreement.

It must be the case that p has all slots up to and including i populated in its log. Furthermore, since t is the first time at which disagreement happens, and p's FUO was at $i-1$ before t, it must be the case that for all values in slots 1 to $i-2$ of p's log, if any other process p' also has those slots committed, then it has the same values as p in those slots. Let p's value at slot $i-1$ be v. Let ℓ_1 be the leader that populated slot $i-1$ for p, and let ℓ_2 be the leader the populated slot i for p. If $\ell_1 = \ell_2$, then p's entry at $i-1$ must be committed at ℓ_1 before time t, since otherwise ℓ_1 would not have started replicating entry i. So, if at time t, some process q has a committed value v' in slot $i-1$ where $v' \neq v$, then this would have violated agreement with ℓ_1 before t, contradicting the assumption that t is the earliest time at which agreement is broken.

Now consider the case where $\ell_1 \neq \ell_2$. Note that for ℓ_2 to replicate an entry at index i, it must have a value v' committed at entry $i-1$. Consider the last leader, ℓ_3, who wrote a value on ℓ_2's $i-1$th entry. If $\ell_3 = \ell_1$, then $v' = v$, since a single leader only ever writes one value on each index. Thus, if agreement is broken by p at time t, then it must have also been broken at an earlier time by ℓ_2, which had v committed at $i-1$ before time t. Contradiction.

If $\ell_3 = \ell_2$, we consider two cases, depending on whether or not p is part of ℓ_2's confirmed followers set. If p is not in the confirmed followers of ℓ_2, then ℓ_2 could not have written a value on p's ith log slot. Therefore, p must have been a confirmed follower of ℓ_2. If p was part of ℓ_2's quorum for committing entry $i-1$, then ℓ_2 was the last leader to write p's $i-1$th slot, contradicting the assumption that ℓ_1 wrote it last. Otherwise, if ℓ_2 did not use p as part of its quorum for committing, it still must have created a work request to write on p's $i-1$th entry before creating the work request to write on p's ith entry. By the FIFOness of RDMA queue pairs, p's $i-1$th slot must therefore have been written by ℓ_2 before the ith slot was written by ℓ_2, leading again to a contradiction.

Finally, consider the case where $\ell_3 \neq \ell_1$ and $\ell_3 \neq \ell_2$. Recall from the previous case that p must be in ℓ_2's confirmed followers set. Then when ℓ_2 takes over as leader, it executes the update followers optimization presented in Algorithm B.2. By executing this, it must update p with its own committed value at $i-1$, and update p's FUO to i. However, this contradicts the assumption that p's FUO was changed from $i-1$ to i by p itself using Algorithm B.3. □

B.4.4 Grow Confirmed Followers

In our algorithm, the leader only writes to and reads from its confirmed followers set. So far, for a given leader ℓ, this set is fixed and does not change after ℓ initially constructs it in lines 4–7. This implies that processes outside of ℓ's confirmed followers set will remain behind and miss updates, even if they are alive and timely.

We present an extension which allows such processes to join ℓ's confirmed followers set even

if they are not part of the initial majority. Every time Propose is invoked, ℓ will check to see if it received permission acks since the last Propose call and if so, will add the corresponding processes to its confirmed followers set. This extension is compatible with those presented in the previous subsections: every time ℓ's confirmed followers set grows, ℓ re-updates its own log from the new followers that joined (in case any of their logs is ahead of ℓ's), as well as updates the new followers' logs (in case any of their logs is behind ℓ's).

One complication raised by this extension is that, if the number of confirmed followers is larger than a majority, then ℓ can no longer wait for its reads and writes to complete at all of its confirmed followers before continuing execution, since that would interfere with termination in an asynchronous system.

The solution is for the leader to issue reads and writes to all of its confirmed followers, but only wait for completion at a majority of them. One crucial observation about this solution is that confirmed followers cannot miss operations or have operations applied out-of-order, even if they are not consistently part of the majority that the leader waits for before continuing. This is due to RDMA's FIFO semantics.

The correctness of this extension derives from the correctness of the algorithm in general; whenever a leader ℓ adds some set S to its confirmed followers C, forming $C' = C \cup S$, the behavior is the same as if ℓ just became leader and its initial confirmed followers set was C'.

B.4.5 Omit Prepare Phase

As is standard practice for Paxos-derived implementations, the prepare phase can be omitted if there is no contention. More specifically, the leader executes the prepare phase until it finds no accepted values during its prepare phase (i.e., until the check at line 26 succeeds). Afterwards, the leader omits the prepare phase until it either (a) aborts, or (b) grows its confirmed followers set; after (a) or (b), the leader executes the prepare phase until the check at line 26 succeeds again, and so on.

This optimization concerns performance on the common path. With this optimization, the cost of a Propose call becomes a single RDMA write to a majority in the common case when there is a single leader.

The correctness of this optimization follows from the following lemma, which states that no 'holes' can form in the log of any replica. That is, if there is a value written in slot i of process p's log, then every slot $j < i$ in p's log has a value written in it.

Lemma B.4.1 (No holes)**.** *For any process p, if p's log contains a value at index i, then p's log contains a value at every index j, $0 \le j \le i$.*

Proof. Assume by contradiction that the lemma does not hold. Let p be a process whose slot j is empty, but slot $j+1$ has a value, for some j. Let ℓ be the leader that wrote the value on slot

$j+1$ of p's log, and let t be the last time at which ℓ gained write permission to p's log before writing the value in slot $j+1$. Note that after time t and as long as ℓ is still leader, p is in ℓ's confirmed followers set. By Algorithm B.2, ℓ must have copied a value into all slots of p that were after p's FUO and before ℓ's FUO. By the way FUO is updated, p's FUO cannot be past slot j at this time. Therefore, if ℓ's FUO is past j, slot j would have been populated by ℓ at this point in time. Otherwise, ℓ starts replicating values to all its confirmed followers, starting at its FUO, which we know is less than or equal to j. By the FIFO order of RDMA queue pairs, p cannot have missed updates written by ℓ. Therefore, since p's $j+1$th slot gets updated by ℓ, so must its jth slot. Contradiction. □

Corollary B.4.2. *Once a leader reads no accepted values from a majority of the followers at slot i, it may safely skip the prepare phase for slots $j > i$ as long as its confirmed followers set does not decrease to less than a majority.*

Proof. Let ℓ be a leader and C be its confirmed follower set which is a quorum. Assume that ℓ executes line 27 for slot i; that is, no follower $p \in C$ had any value in slot i. Then, by Lemma B.4.1, no follower in C has any value for any slot $j > i$. Since this constitutes a majority of the processes, no value is decided in any slot $j > i$, and by Invariant B.3.1, no value is committed at any process at any slot $j > i$. Furthermore, as long as ℓ has the write permission at a majority of the processes, ℓ is the only one that can commit new entries in these slots (by Invariant B.3.5). Thus, ℓ cannot break agreement by skipping the prepare phase on the processes in its confirm followers set. □

C The Inherent Cost of Remembering Consistently

Corner Case: Reader Traversal Concurrent with Update

In Figure C.1 we exemplify a read from a state that is never the latest in the non-fuzzy part. In part A, a reader starts and finds that the available flag of Node 4 is still unset. It continue with Node 3, but scheduled out before checking its available flag. The non-fuzzy part consist of INIT,op1,op2. In part B, the thread executing op4 (Thread 4) sets the available flag of Node 4. The non-fuzzy part is expanded to INIT,op1,op2,op3,op4. We emphasize that Node 3 is part of the non-fuzzy part and that op3 is already linearized. In part C, the thread executing op3 (Thread 3) sets the available flag of Node 3. This is a redundant operation with respect to op3, which was already linearized at part B; but Thread 3 is unaware of op4 and continues with the update algorithm. In part D, the reader resume, finds that the available flag of op3 is set, and computes the returned value based on the state up to op3 (INIT,op1,op2,op3). This state was never the non-fuzzy part of the execution trace.

The read operation is still correctly linearized since moving from history INIT,op1,op2 to history INIT,op1,op2,op3,op4 must pass through history INIT,op1,op2,op3. This is the linearization point of the read. Using the terminology of Lemma 4.5.7, op3 is linearized at time $t_4 - \epsilon$ and the reader is linearized between $t_4 - \epsilon$ and t_4.

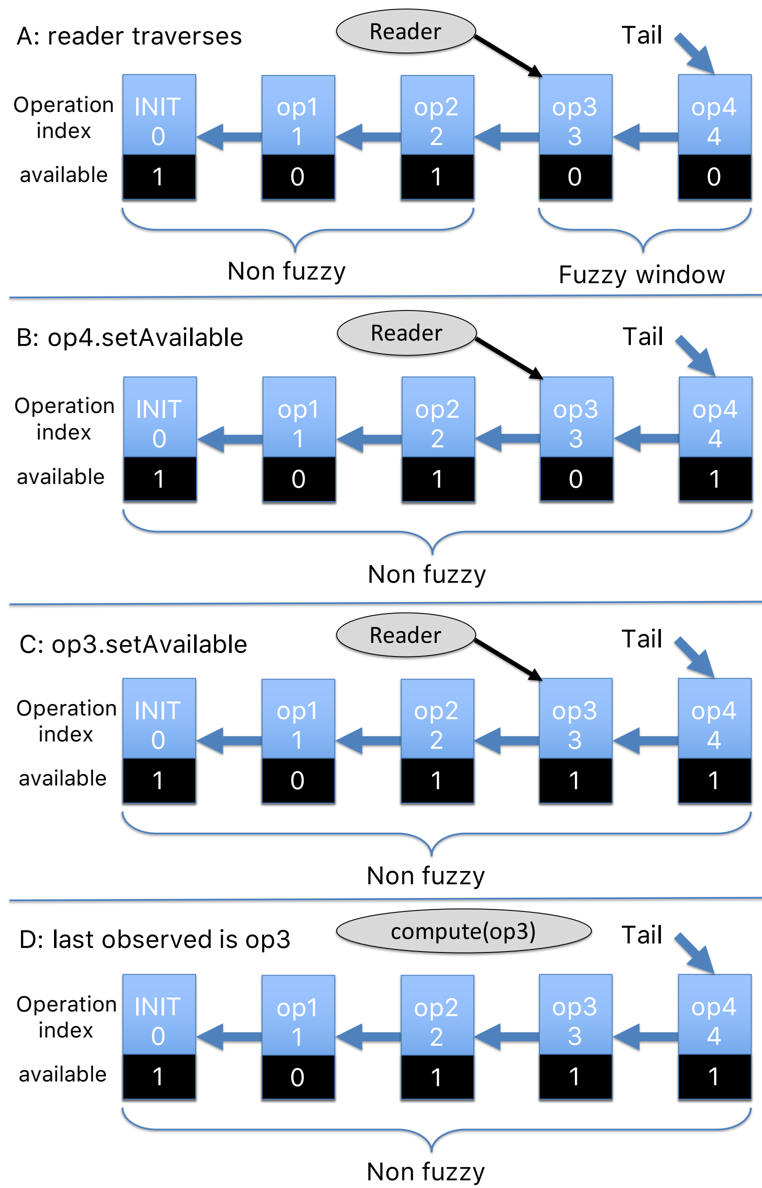


Figure C.1 – Illustrating a read from a node that is never the latest in the non-fuzzy part.

D Efficient Multi-Word Compare-and-Swap

D.1 Replacing RDCSS with CAS in Harris et al. algorithm leads to ABA

Consider two memory locations a_1 and a_2 with initial values v_1 and v_2 respectively. Consider two 2-CAS operations op and op' which operate on a_1 and a_2. op has old values v_1 and v_2 and new values v'_1 and v'_2, respectively; op' has old values v'_1 and v'_2 and new values v_1 and v_2, respectively. Let D be op's descriptor.

1. op executes solo and performs the CAS to make a_1 point to D, then pauses immediately before the CAS to acquire a_2.
2. op' executes solo: it first helps op complete, changing the values of a_1 and a_2 to v'_1 and v'_2 respectively; then op' performs its own changes, modifying a_1's and a_2's values back to v_1 and v_2, respectively.
3. op resumes, successfully acquires a_2, performs the status-change CAS on D, then performs unlocking CASes on a_1 and a_2. The CAS on a_1 will fail, and a_1's value will remain v_1. The CAS on a_2 will succeed, changing its value to v'_2.
4. The values of a_1 and a_2 are now incompatible with any linearization of op and op'.

D.2 Performance in read-only and update-heavy workloads

Figures D.1 and D.2 show performance results for the 50% reads and 100% reads workloads in the doubly-linked list and B+-tree benchmarks, respectively. All other settings are the same as in Section 5.7. These more extreme workloads largely magnify the performance effects demonstrated by the workloads included in Section 5.7. Namely, as the write ratio increases, so does the gap by which AOPT outperforms PMwCAS in cases that involve contention between concurrent MCAS operations (i.e., small and moderate lists and trees). With 100% reads, our

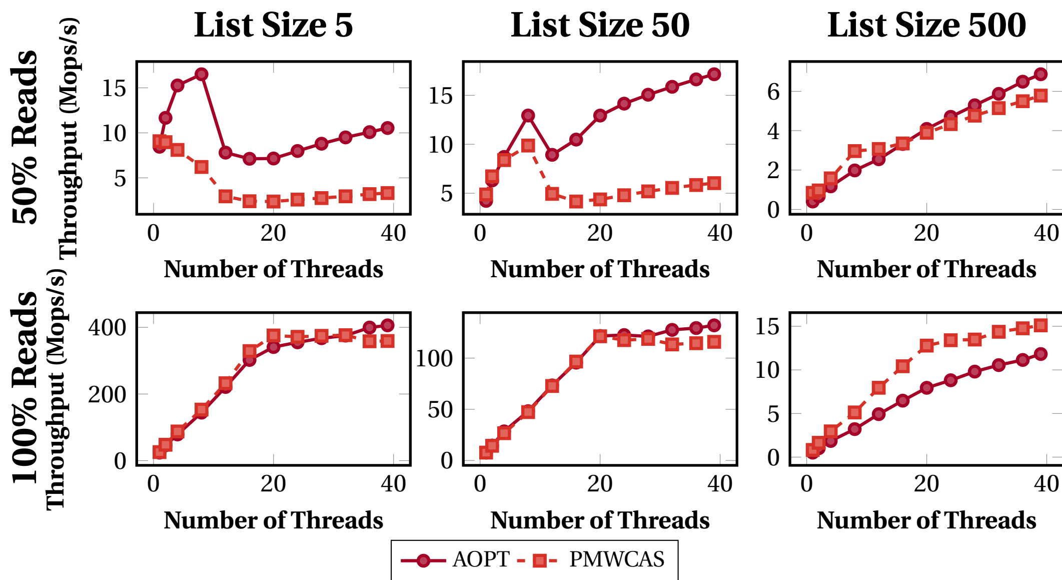


Figure D.1 – Doubly-linked list benchmark. Top row shows results for 50% reads workload; bottom row shows results for 100% reads workload. Each column corresponds to a different initial list size (5, 50 and 500 elements, respectively).

algorithm performs on par with or slightly trails behind PMwCAS at every contention level. Since the workload involves no update operations (and thus no MCASes), the lower complexity of our MCAS operations does not factor into the results, whereas the higher overhead of the extra-level of indirection in our read operations does factor in.

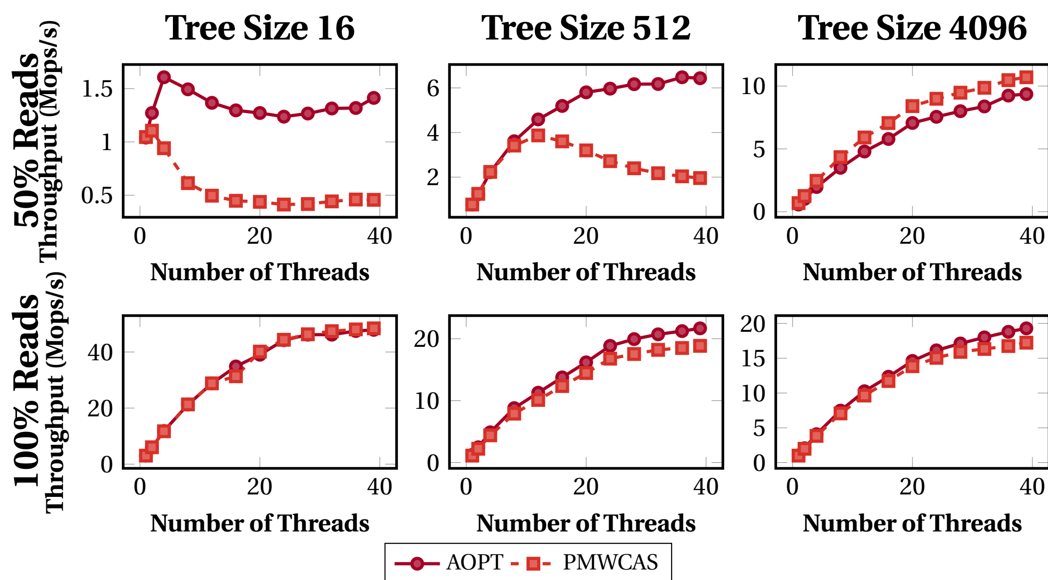


Figure D.2 – B+-tree benchmark. Top row shows results for 50% reads workload; bottom row shows results for 100% reads workload. Each column corresponds to a different initial tree size (16, 512 and 4096 elements, respectively).

E Fast and Robust Memory Reclamation for Concurrent Data Structures

E.1 QSBR Correctness

Lemma E.1.1. *If the time interval $[a, b]$ is a grace period, after time b no process holds hazardous references to nodes that were removed before time a.*

Proof. Let $[a, b]$ be a grace period and consider a node n that was removed at time $t < a$. By the definition of the removed state, this means that no process can obtain a reference to n after a, but it is possible for processes to still hold references to n that they obtained before t. By the definition of a grace period, all processes will pass through a quiescent state between a and b. Therefore, for each process p there exists a time t_j, $a \leq t_j \leq b$, when p does not hold any reference to n. Thus, at b, none of the processes hold any references (and in particular, hazardous references) to n. □

Lemma E.1.2. *For any process p, at the time when p updates its local epoch from e_j to e_G, no process holds any hazardous references to the nodes already present in p's ${e_G}^{th}$ limbo list.*

Proof. Note that all epoch updates are done modulo three (because there are three logical epochs). Without loss of generality, suppose process p passes through a local epoch cycle $0 \to 1 \to 2 \to 0$. We want to show that when p reaches epoch 0 for the second time, no processes hold any hazardous references to the nodes in p's limbo list 0.

We claim that the system goes through a grace period $[a, b]$ starting just before the quiescent state during which p updates its local epoch from 0 to 1 and ending just after the transition by p of its local epoch from 2 to 0. Note that this claim implies, using Lemma E.1.1, that after p's local epoch becomes 0 again, no processes hold hazardous references to the nodes in p's limbo list 0, as required to complete the proof of Lemma E.1.2.

We now proceed to prove the claim. Assume that $[a, b]$ is not a grace period. It follows that there exists a process q that does not go through a quiescent state during $[a, b]$. Therefore, we know that the local epoch of q stays the same during the time interval $[a, b]$. Since during

$[a, b]$, p updates its local epoch from 0 to 1, then from 1 to 2 and then from 2 to 0, there exist times $t_1 < t_2 < t_3$, $t_1, t_2, t_3 \in [a, b]$ such that $e_G = 1$ at t_1, $e_G = 2$ at t_2 and $e_G = 0$ at t_3. Since some process transitions the global epoch from 1 to 2 between t_1 and t_2, it must be the case the the epoch of q is equal to 1 (otherwise the update cannot be completed). But this means that later during $[a, b]$ the global epoch cannot be advanced from 2 to 0, because there exists at least one process (q) whose local epoch is not equal to 2. We have reached a contradiction. This completes the proof of the claim and of Lemma E.1.2 . □

Property E.1.3 (Safety)**.** *If at time t, node n is identified by process p as eligible for reuse, then no process holds any hazardous references to n at time t.*

Proof. This follows from Lemma E.1.2 and from the fact that a process p will identify a node n as eligible for reuse if and only if p has just updated its local epoch from e_j to e_G and n is in p's $e_G{}^{\text{th}}$ limbo list. □

E.2 QSense on a Linked List

Algorithms E.1, E.2 and E.3 show an example of how QSense can be applied to a lock-free concurrent linked-list [104]. The lines of code needed to use QSense are highlighted (in blue). First, in the beginning of each list operation, the `manage_qsense_state` function is called. This function takes care of switching between QSBR and Cadence, if necessary, and invoking a quiescent state for every batch of performed operations. Second, hazard pointers are assigned to protect nodes when the list is traversed, in the same way one would use the original hazard pointer technique. The only difference is that the memory barrier between the hazard pointer assignment and the verification step is no longer needed. Finally, `free_node_later` should be called instead of `free`, when a node is removed.

Algorithm E.1 – QSense on a concurrent linked-list (I).

```
1  Node* search(Node* set_head, Key key) {
2      manage_qsense_state();
3      Node *left_node, *right_node;
4  retry_search:
5      left_node = set_head;
6      right_node = set_head->next;
7      while (True) {
8          //Protect node by hazard pointer and perform verification,
             ↪ without the memory barrier
9          assign_HP(left_node, 0); assign_HP(right_node, 1);
10         if (right_node != left_node->next) {
11             goto retry_search;
12         }
13         if (right_node->key ≥ key) {
14             break;
15         }
16         left_node = right_node;
17         right_node = unmarked(right_node->next);
18     }
19     return right_node;
20 }

22 Boolean insert(Node *list_head, Key key) {
23     manage_qsense_state();
24     do {
25         Node* left_node;
26         Node* right_node = search_and_cleanup(list_head, key, &left_node)
             ↪ ;
27         if (right_node->key == key) {
28             return False;
29         }
30         //Allocate a node with the allocator of your choice
31         Node* node_to_add = new_node(key, right_node);
32         if (CAS(&left_node->next, right_node, node_to_add) == right_node)
             ↪ {
33             return True;
34         }
35         //Node was not inserted; free the node directly.
36         free(node_to_add);
37     } while (True);
38 }
```

Algorithm E.2 – QSense on a concurrent linked-list (II).

```
39 Boolean delete(Node *list_head, Key key) {
40     manage_qsense_state();
41     Node* cas_result;
42     Node* unmarked_node;
43     Node* left_node;
44     Node* right_node;
45     do {
46         right_node = search_and_cleanup(list_head, key, &left_node);
47         if (right_node->key != key) {
48             return False;
49         }
50         //Try to mark right_node as logically deleted
51         unmarked_node = unmarked(right_node->next);
52         Node* marked_node = marked(unmarked_node);
53         cas_result = CAS(&right_node->next, unmarked_node, marked_node);
54     } while (cas_result != unmarked_node);
55     if (!unlink_right(left_node, right_node)) {
56         search_and_cleanup(list_head, key, &left_node);
57     }
58     return True;
59 }

61 Boolean unlink_right(Node* left_node, Node* right_node) {
62     Node* new_next = unmarked(right_node->next);
63     Node* old_right_node = CAS(&left_node->next, right_node, new_next);
64     Boolean removed = (old_right_node == right_node);
65     if (removed){
66         //call instead of free
67         free_node_later(old_right_node);
68     }
69     return removed;
70 }
```

Algorithm E.3 – QSense on a concurrent linked-list (III).

```
71  Node* search_and_cleanup(Node* set_head, Key key, Node** left_node_ref) {
72      Node *left_node, *right_node;
73  retry_search_cleanup:
74      left_node = set_head;
75      right_node = set_head->next;
76      while (True) {
77          //Protect node by hazard pointer and perform verification,
              ↪ without the memory barrier
78          assign_HP(left_node, 0); assign_HP(right_node, 1);
79          if (right_node != left_node->next) {
80              goto retry_search_cleanup;
81          }
82          if (!is_marked(right_node->next)) {
83              if (right_node->key ≥ key) {
84                  break;
85              }
86              left_node = right_node;
87          } else {
88              //Perform cleanup of marked node
89              unlink_right(left_node, right_node);
90          }
91          right_node = unmarked(right_node->next);
92      }
93      *left_node_ref = left_node;
94      return right_node;
95  }
```

Bibliography

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed computing (DIST)*, 18(5), 2006.

[2] Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. In *ACM International Symposium on Memory Management (ISMM)*, 2011.

[3] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1992.

[4] Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2014.

[5] Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2018.

[6] Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, 2003.

[7] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. StackTrack: An automated transactional approach to concurrent memory reclamation. In *European Conference on Computer Systems (EuroSys)*, 2014.

[8] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[9] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *European Conference on Computer Systems (EuroSys)*, 2017.

[10] Noga Alon, Michael Merritt, Omer Reingold, Gadi Taubenfeld, and Rebecca N Wright. Tight bounds for shared memory systems accessed by Byzantine processes. *Distributed computing (DIST)*, 18(2), 2005.

[11] James H. Anderson and Mark Moir. Universal Constructions for Multi-object Operations. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[12] James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing Wait-free Objects on Priority-based Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1997.

[13] Anonymous. 1588-2019—ieee approved draft standard for a precision clock synchronization protocol for networked measurement and control systems. https://standards.ieee.org/content/ieee-standards/en/standard/1588-2019.html.

[14] Order matching system. https://en.wikipedia.org/wiki/Order_matching_system.

[15] Anonymous. When microseconds count: Fast current loop innovation helps motors work smarter, not harder. http://e2e.ti.com/blogs_/b/thinkinnovate/archive/2017/11/14/when-microseconds-count-fast-current-loop-innovation-helps-motors-work-smarter-not-harder.

[16] Andrea Arcangeli, Mingming Cao, Paul E McKenney, and Dipankar Sarma. Using read-copy-update techniques for System V IPC in the Linux 2.5 kernel. In *USENIX Annual Technical Conference (ATC)*, 2003.

[17] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. In *International Conference on Very Large Databases (VLDB)*, 2018.

[18] ASCYLIB, a concurrent-search data-structure library with over 40 implementations of linked lists, hash tables, skip lists, binary search trees, queues, and stacks. https://github.com/LPD-EPFL/ASCYLIB, 2018.

[19] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3), 1990.

[20] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1), 1995.

[21] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2011.

[22] Hagit Attiya and Eshcar Hillel. Highly concurrent multi-word synchronization. *Theoretical Computer Science*, 412(12-14), 2011.

[23] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4), 2015.

[24] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A Distributed Shared Log. In *ACM Transactions on Computer Systems (TOCS)*, 2013.

[25] Mahesh Balakrishnan, Aviad Zuck, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, and Tao Zou. Tango: Distributed Data Structures Over a Shared Log. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[26] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, April 2017.

[27] Rida Bazzi and Gil Neiger. Optimally simulating crash failures in a Byzantine environment. In *International Workshop on Distributed Algorithms (WDAG)*, 1991.

[28] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1983.

[29] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2015.

[30] Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2015.

[31] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3), 2009.

[32] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1), 1987.

[33] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence Programming Models for Non-volatile Memory. In *ACM International Symposium on Memory Management (ISMM)*, 2016.

[34] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34(1):47–67, 2003.

[35] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Reconstructing Paxos. *ACM SIGACT News*, 34(2), 2003.

[36] Sol Boucher, Anuj Kalia, and David G. Andersen. Putting the “micro” back in microservice. In *USENIX Annual Technical Conference (ATC)*, 2018.

[37] Zohir Bouzid, Damien Imbs, and Michel Raynal. A necessary condition for Byzantine k-set agreement. *Information Processing Letters*, 116(12), 2016.

[38] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2), 1987.

[39] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4), 1985.

[40] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[41] Anastasia Braginsky and Erez Petrank. A Lock-free B+Tree. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[42] Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *International Conference on Parallel Computing Technologies*, 2001.

[43] Trevor Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.

[44] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic Primitives for Non-blocking Data Structures. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2013.

[45] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[46] Bztree: a high-performance latch-free range index for non-volatile memory. https://github.com/wangtzh/bztree, 2019.

[47] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer, 2011.

[48] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[49] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1999.

[50] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3), August 2003.

[51] Diego Cepeda, Sakib Chowdhury, Nan Li, Raphael Lopez, Xinzhe Wang, and Wojciech Golab. Toward linearizability testing for multi-word persistent synchronization primitives. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2019.

[52] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[53] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[54] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2), 1996.

[55] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery Write-ahead System for In-memory Non-volatile Data-structures. In *International Conference on Very Large Databases (VLDB)*, 2015.

[56] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMove: Helping Programmers Move to Byte-Based Persistence. In *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2016.

[57] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. In *International Conference on Very Large Databases (VLDB)*, 2015.

[58] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine Byzantine-fault tolerance. In *USENIX Annual Technical Conference (ATC)*, 2008.

[59] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[60] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2012.

[61] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[62] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[63] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[64] Nachshon Cohen, Michal Friedman, and James R. Larus. Efficient Logging in Non-volatile Memory by Exploiting Coherency Protocols. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.

[65] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The Inherent Cost of Remembering Consistently. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[66] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[67] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[68] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *International Symposium on Reliable Distributed Systems (SRDS)*, 2004.

[69] Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous Byzantine consensus with $2f+1$ processes. In *ACM symposium on applied computing (SAC)*, 2010.

[70] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-driven tail-aware balancing of μs-scale RPCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[71] Al Danial. cloc: Count lines of code. https://github.com/AlDanial/cloc.

[72] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *USENIX Annual Technical Conference (ATC)*, 2018.

[73] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[74] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[75] Joel Edward Denny, Seyong Lee, and Jeffrey S. Vetter. Language-Based Optimizations for Persistence on Nonvolatile Main Memory Systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[76] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. Even Better DCAS-Based Concurrent Deques. In *International Symposium on Distributed Computing (DISC)*, 2000.

[77] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

[78] Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *ACM International Symposium on Memory Management (ISMM)*, 2016.

[79] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[80] Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *International Conference on Dependable Systems and Networks (DSN)*, 2006.

[81] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. DCAS is Not a Silver Bullet for Nonblocking Algorithm Design. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2004.

[82] Travis Downs. A benchmark for low-level cpu micro-architectural features. https://github.com/travisdowns/uarch-bench.

[83] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2011.

[84] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[85] Aleksandar Dragojevic, Dushyanth Narayanan, Ed Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[86] Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *International Conference on Dependable Systems and Networks (DSN)*, 2005.

[87] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2), 1988.

[88] Steven D. Feldman, Pierre LaBorde, and Damian Dechev. A Wait-Free Multi-Word Compare-and-Swap Operation. *International Journal of Parallel Programming*, 43(4), 2015.

[89] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 1985.

[90] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[91] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[92] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed computing (DIST)*, 16(1), 2003.

[93] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, April 2019.

[94] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8), 2008.

[95] Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic Scalable Atomicity via Semantic Locking. *ACM Transactions on Parallel Computing*, 3(4), 2017.

[96] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. Using Storage Class Memory Efficiently for an In-memory Database. In *ACM International Systems and Storage Conference (SYSTOR)*, 2016.

[97] James N Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, volume 60, pages 393–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978.

[98] Michael Greenwald. *Non-blocking synchronization and system design*. PhD thesis, Stanford University, 1999.

[99] Michael Greenwald. Two-handed emulation: how to build non-blocking implementation of complex data-structures using DCAS. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[100] Rachid Guerraoui and Ron R Levy. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[101] Phuong Hoai Ha and Philippas Tsigas. Reactive Multi-Word Synchronization for Multiprocessors. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2003.

[102] Phuong Hoai Ha and Philippas Tsigas. Reactive Multi-word Synchronization for Multiprocessors. *Journal of Instruction-Level Parallelism*, 6, 2004.

[103] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory: 2nd Edition*. Morgan & Claypool, 2010.

[104] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked Lists. In *International Symposium on Distributed Computing (DISC)*, 2001.

[105] Timothy L. Harris and Keir Fraser. Language support for lightweight transactions. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[106] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *International Symposium on Distributed Computing (DISC)*, 2002.

[107] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12), 2007.

[108] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[109] Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1), 1991.

[110] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5), 1993.

[111] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2), 2005.

[112] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *International Symposium on Distributed Computing (DISC)*, 2002.

[113] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

[114] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)*, 1993.

[115] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., 1st edition, 2012.

[116] Maurice Herlihy and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[117] Brandon Holt, James Bornholt, Irene Zhang, Dan R. K. Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Symposium on Cloud Computing (SoCC)*, 2016.

[118] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *European Conference on Computer Systems (EuroSys)*, 2017.

[119] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[120] Intel. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf.

[121] Intel. Intel64 and IA-32 Architectures Optimization Reference Manual. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[122] Intel. Intel64 and IA-32 Architectures Software Developers Manuals Combined. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[123] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2018.

[124] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

[125] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[126] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic Persistent Memory Updates via JUSTDO Logging. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[127] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *International Symposium on Distributed Computing (DISC)*, 2016.

[128] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3), 1998.

[129] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2), 2019.

[130] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[131] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[132] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4), 2015.

[133] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, 2016.

[134] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2016.

[135] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *European Conference on Computer Systems (EuroSys)*, 2012.

[136] Antonios Katsarakis, Vasilis Avrielatos, M R Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojević, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[137] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2), 2001.

[138] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-ahead Logging. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[139] Alex Kogan and Erez Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[140] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *European Conference on Computer Systems (EuroSys)*, 2020.

[141] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance Transactions for Persistent Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[142] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[143] Klaus Kursawe. Optimistic Byzantine agreement. In *International Symposium on Reliable Distributed Systems (SRDS)*, 2002.

[144] Edya Ladan-Mozes, I-Ting Angelina Lee, and Dmitry Vyukov. Location-based memory fences. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[145] Leslie Lamport. The weak Byzantine generals problem. *Journal of the ACM (JACM)*, 30(3), 1983.

[146] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

[147] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), 2001.

[148] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[149] Leslie Lamport. Fast Paxos. *Distributed computing (DIST)*, 19(2), 2006.

[150] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

[151] Butler W Lampson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms (WDAG)*, 1996.

[152] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[153] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[154] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[155] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *European Conference on Computer Systems (EuroSys)*, 2015.

[156] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[157] Mohsen Lesani, Todd Millstein, and Jens Palsberg. Automatic Atomicity Verification for Clients of Concurrent Data Structures. In *International Conference on Computer Aided Verification (CAV)*, 2014.

[158] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *IEEE International Conference on Data Engineering (ICDE)*, 2013.

[159] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: datacenter sockets can be fast and compatible. In *ACM Conference on SIGCOMM*, 2019.

[160] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2016.

[161] Liquibook. https://github.com/enewhuis/liquibook. Accessed 2020-05-25.

[162] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3), 2004.

[163] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DUDETM: Building Durable Transactions with Decoupling for Persistent Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[164] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *ACM Conference on SIGCOMM*, 2019.

[165] Robert Love. *Linux System Programming: Talking Directly to the Kernel and C Library, 2nd Edition.* O'Reilly Media, Inc., 2013.

[166] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2003.

[167] Dahlia Malkhi, Michael Merritt, Michael K Reiter, and Gadi Taubenfeld. Objects shared by Byzantine processes. *Distributed computing (DIST)*, 16(1), 2003.

[168] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(3), 2006.

[169] David Mazieres. Paxos made practical. https://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, 2007.

[170] Paul E McKenney. Memory barriers: a hardware view for software hackers, 2010.

[171] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it?, 2017.

[172] Paul E McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Ottawa Linux Symposium*, 2002.

[173] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 1998.

[174] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 1998.

[175] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *European Conference on Computer Systems (EuroSys)*, 2017.

[176] Memcached. http://www.memcached.org.

[177] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2002.

[178] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 2004.

[179] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 2004.

[180] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[181] Micron. 3D XPoint Technbology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[182] Mark Moir. Transparent Support for Wait-Free Transactions. In *International Workshop on Distributed Algorithms (WDAG)*, 1997.

[183] Adam Morrison and Yehuda Afek. Temporally bounding TSO for fence-free asymmetric synchronization. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[184] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[185] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3), 1990.

[186] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *ACM International Systems and Storage Conference (SYSTOR)*, 2019.

[187] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[188] The OpenMP API specification for parallel programming. https://www.openmp.org/, 2019.

[189] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-tree for Storage Class Memory. In *International Conference on Management of Data (SIGMOD)*, 2016.

[190] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[191] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-TSO. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[192] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[193] Matej Pavlovic, Alex Kogan, Virendra J. Marathe, and Tim Harris. Persistent Multi-Word Compare-and-Swap. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2018.

[194] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2), 1980.

[195] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed computing (DIST)*, 15(2), 2002.

[196] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3), 2015.

[197] Benchmarking framework for index structures on persistent memory. https://github.com/wangtzh/pibench, 2019.

[198] Persistent multi-word compare-and-swap (PMwCAS) for NVRAM. https://github.com/microsoft/pmwcas, 2019.

[199] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on RDMA networks. In *Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2015.

[200] Manuel Pöter and Jesper Larsson Träff. *Stamp-it*, amortized constant-time memory reclamation in comparison to five other schemes. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[201] Mary C. Potter, Brad Wyble, Carl Erick Hagmann, and Emily Sarah McCourt. Detecting meaning in RSVP at 13 ms per picture. *Attention, Perception, & Psychophysics*, 76(2):270–279, 2014.

[202] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[203] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Symposium on Cloud Computing (SoCC)*, 2015.

[204] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2018.

[205] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[206] Redis. https://redis.io/. Accessed 2020-05-25.

[207] David Reinsel, John Gantz, and John Rydning. The digitization of the world from edge to core, 2018. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[208] Stephen Mathew Rumble. *Memory and object management in RAMCloud*. PhD thesis, Stanford University, 2014.

[209] Signe Rüsch, Ines Messadi, and Rüdiger Kapitza. Towards low-latency Byzantine agreement protocols using RDMA. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018.

[210] David Schneider. The microsecond market. *IEEE Spectrum*, 49(6), June 2012.

[211] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[212] Ohad Shacham, Eran Yahav, Guy Golan Gueta, Alex Aiken, Nathan Bronson, Mooly Sagiv, and Martin Vechev. Verifying Atomicity via Data Independence. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2014.

[213] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, October 2011.

[214] Nir Shavit and Dan Touitou. Software Transactional Memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[215] Dale Skeen. Nonblocking commit protocols. In *International Conference on Management of Data (SIGMOD)*, pages 133–142, 1981.

[216] SNIA. Extending RDMA for persistent memory over fabrics. https://www.snia.org/sites/default/files/ESF/Extending-RDMA-for-Persistent-Memory-over-Fabrics-Final.pdf.

[217] Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *International Symposium on Distributed Computing (DISC)*, 2008.

[218] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191), 2008.

[219] Håkan Sundell. Wait-Free Multi-Word Compare-and-Swap Using Greedy Helping and Grabbing. *International Journal of Parallel Programming*, 39(6), 2011.

[220] Shahar Timnat, Maurice Herlihy, and Erez Petrank. A Practical Transactional Memory Interface. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2015.

[221] Shahar Timnat and Erez Petrank. A Practical Wait-free Simulation for Lock-free Data Structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[222] John D. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[223] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1), 2013.

[224] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight Persistent Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[225] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable Paxos on RDMA. In *Symposium on Cloud Computing (SoCC)*, 2017.

[226] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *IEEE International Conference on Data Engineering (ICDE)*, 2018.

[227] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[228] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[229] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[230] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

Curriculum Vitae

Igor Zabloțchi

E-mail: igor.zablotchi@epfl.ch
Phone: +41 78 941 92 36

Research Areas

state-machine replication, Byzantine fault tolerance, RDMA, non-volatile memory, concurrent data structures, concurrent memory reclamation

Education

2020 PhD in Computer Science, EPFL
Thesis: *Distributed Computing with Modern Shared Memory*
Advisor: Rachid Guerraoui

2015 MSc in Computer Science, EPFL
Thesis: *SMR-NoMembar — Eliminating Memory Barriers from Hazard Pointers*
Supervisors: Maurice Herlihy, Rachid Guerraoui
GPA: 5.91/6 — Ranked 2/89 in CS Section

2012 BSc in Computer Science, EPFL
Thesis: *Crossing Flocks of Agents Using Virtual Leaders*
Supervisor: Denis Gillet
GPA: 5.83/6 — Ranked 1/77 in CS Section and 3/769 in EPFL overall

Internships & Exchanges

2019 Microsoft Research, Cambridge, UK — Research Internship
- Supervised by Aleksandar Dragojević
- Topic: hardware-software concurrent data structures
- Programming language: C++

2018 Oracle Labs, Burlington, MA — Research Internship
- Supervised by Virendra Marathe
- Topic: fast RDMA-based consensus protocols, efficient multi-word compare-and-swap
- Programming language: C++

2016 VMware Research, Palo Alto, CA — Research Internship
- Supervised by Dahlia Malkhi and Ittai Abraham
- Topic: transactions across blockchain ledgers with atomicity, fairness and expressiveness

2015 EPFL, Distributed Programming Laboratory — Research Internship
- Supervised by Vasileios Trigonakis and Rachid Guerraoui
- Topic: fast concurrent persistent key-value store
- Programming languages: C, C++

2014–2015 Brown University — MSc Thesis Project (exchange semester)
- Supervised by Maurice Herlihy
- Topic: fast and robust concurrent memory reclamation
- Programming language: C

2013 ABB Research Switzerland — Research Internship
- Supervised by Ettore Ferranti and Yvonne-Anne Pignolet
- Topics: building automation, brain-computer interface, domain-specific languages
- Languages & tools: Python, Matlab, Xtext, OpenVIBE, Emotiv EPOC

Research Output

Peer-reviewed Conference Papers [author names in alphabetical order]

2019 *The Impact of RDMA on Agreement*. PODC '19.
Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe and **Igor Zablotchi**.

2018 *Log-Free Concurrent Data Structures*. USENIX ATC '18.
Tudor David, Aleksandar Dragojević, Rachid Guerraoui and **Igor Zablotchi**.

2018 *The Inherent Cost of Remembering Consistently*. SPAA '18.
Nachshon Cohen, Rachid Guerraoui and **Igor Zablotchi**

2017 *FloDB: Unlocking Memory in Persistent Key-Value Stores*. EuroSys '17.
Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis and **Igor Zablotchi**.

2017 *The Disclosure Power of Shared Objects*. NETYS '17.
Peva Blanchard, Rachid Guerraoui, Julien Stainer and **Igor Zablotchi**.

2016 *Fast and Robust Memory Reclamation for Concurrent Data Structures*. SPAA '16.
Oana Balmau, Rachid Guerraoui, Maurice Herlihy and **Igor Zablotchi**.

Conference Papers under Submission [author names in alphabetical order]

2020 *Microsecond Consensus for Microsecond Applications.*
Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis and **Igor Zablotchi**.

2020 *Efficient Multi-word Compare and Swap.*
Rachid Guerraoui, Alex Kogan, Virendra J. Marathe and **Igor Zablotchi**.

2020 *Leaderless Consensus.*
Karolos Antoniadis, Vincent Gramoli, Rachid Guerraoui, Eric Ruppert and **Igor Zablotchi**.

Conference Presentations

2018 The Inherent Cost of Remembering Consistently. SPAA '18

2017 The Disclosure Power of Shared Objects. NETYS '17.

2016 Fast and Robust Memory Reclamation for Concurrent Data Structures. SPAA '16.

Conference Posters

2018 Log-Free Concurrent Data Structures. USENIX ATC '18

2017 FloDB: Unlocking Memory in Persistent Key-Value Stores. EuroSys '17

Languages

English *&* French — fluent • Romanian — native language

Honors & Awards

2019 EPFL IC Teaching Assistant Award
2015 EPFL PhD Fellowship
2015 *Société Suisse d'Informatique* Prize — for achieving 2nd highest GPA in EPFL CS MSc Program
2012 EPFL MSc Excellence Fellowship
2012 EPFL Prize — for achieving 3rd highest GPA in 2012 graduating class

Teaching

Teaching Assistant

2016-2020 Concurrent Algorithms. Graduate class. EPFL
2019 Information Security and Privacy. Graduate class. EPFL
2017 Digital System Design. Undergraduate class. EPFL
2016 Practice of Object-Oriented Programming. Undergraduate class. EPFL

Student Assistant

2014-2015 Natural Language Processing. Graduate class. EPFL
2010-2014 Discrete Mathematics, Calculus, Linear Algebra. Undergraduate classes. EPFL

Mentoring

2019-2020 Loïc Vandenberghe and Manuel Vidigueira. *Fast RDMA Consensus.* MSc Semester Project. EPFL
2018 Ivi Dimopoulou. *Implementation and Evaluation of 1-Fence Concurrent Persistent Data Structures.* MSc Semester Project. EPFL

Professional Service

External Reviewer

2019, 2017 DISC (International Symposium on Distributed Computing)
2017 IPDPS (International Parallel and Distributed Processing Symposium)
2015 SPAA (Symposium on Parallelism in Algorithms and Architectures)